This project has received funding from the European's Union Horizon 2020 research innovation programme under Grant Agreement No. 957258

Architecture for Scalable, Self-human-centric, Intelligent, Secure, and Tactile next generation IoT

D6.7 Release and Distribution Plan – Final

Deliverable No.	D6.7	**Due Date**	30-Apr-2023
Type	Report	**Dissemination Level**	Public
Version	1.0	**WP**	WP6
Description	Software and documentation release and distribution plan to be followed by all enablers. This version will present the final execution outcomes.		

Copyright

The ASSIST-IoT consortium consists of the following 15 partners:

UNIVERSITAT POLITÈCNICA DE VALÈNCIA	Spain
PRODEVELOP S.L.	Spain
SYSTEMS RESEARCH INSTITUTE POLISH ACADEMY OF SCIENCES IBS PAN	Poland
ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS	Greece
TERMINAL LINK SAS	France
INFOLYSIS P.C.	Greece
CENTRALNY INSTYUT OCHRONY PRACY	Poland
MOSTOSTAL WARSZAWA S.A.	Poland
NEWAYS TECHNOLOGIES BV	Netherlands
INSTITUTE OF COMMUNICATION AND COMPUTER SYSTEMS	Greece
KONECRANES FINLAND OY	Finland
FORD-WERKE GMBH	Germany
GRUPO S 21SEC GESTION SA	Spain
TWOTRONIC GMBH	Germany
ORANGE POLSKA SPOLKA AKCYJNA	Poland

Disclaimer

Authors

Name	Partner	e-mail
Alejandro Fornés	P01 UPV	alforlea@upv.es
Rafael Vañó	P01 UPV	ravagar2@upv.es
Juan Gascón	P01 UPV	juagasre@upv.es
Raúl Reinosa	P01 UPV	rreisim@upv.es

History

Date	Version	Change
22-Feb-2023	0.1	ToC and task assignments
31-Mar-2023	0.2	First round of contributions
06-Apr-2023	0.3	Second round of contributions
13-Apr-2023	0.4	Refinement and annex included
24-Apr-2023	0.9	Version from IR received
05-May-2023	1.0	Version released to EC

Key Data

Keywords	Release, software, Helm, packaging, GitLab, DevSecOps
Lead Editor	P01 UPV – Alejandro Fornés
Internal Reviewer(s)	P08 NEWAYS – Johan Schabbink, P14 TWOT – Lambis Tassakos

Executive Summary

This deliverable presents all the actions carried out for implementing the packaging and releasing strategy of the project, focusing on guidelines, tools (custom-made, from third parties) and results. Regarding **packaging**, Helm charts have been used for packaging the enablers of the project, making some **adaptations** over their typical structure and conventions to ease enablers integration in the project. This report delves on these adaptations, presenting the final version of the chart generation to smooth their preparation (and, to adapt existing ones). With respect to **releasing**, the artifacts registries and licensing strategy (leveraging Apache 2.0 or similar permissive licenses) are also updated. Until M30, code, container images and chart packages have been kept primarily on private repositories; however, from now on, **container images and packages** will be hosted in the project's **public** registries, to be released in GitHub, DockerHub and ArtifactHub repositories once the technical developments are finished.

Two **phases** were originally planned for the release and distribution of artifacts. With the project extension, the number of phases has been extended to three. The first phase ended with the submission of D6.4, with a **first** functional version of the essential enablers (M18), being only some of them packaged (some non-essential enablers were also delivered). The **second** phase, concurrent to the present deliverable, includes functional versions of all the enablers, being almost all of them packaged. All this process is supported by an automated **CI/CD pipeline** with dedicated scripts. Finally, the **third** phase will focus mostly on refinements, considering feedback from users (e.g., pilots, open callers). To conclude, instructions for configuring and deploying enablers are provided, in any virtualised system managed by K8s (governed by the Smart orchestrator, or not).

Table of contents

List of tables

List of figures

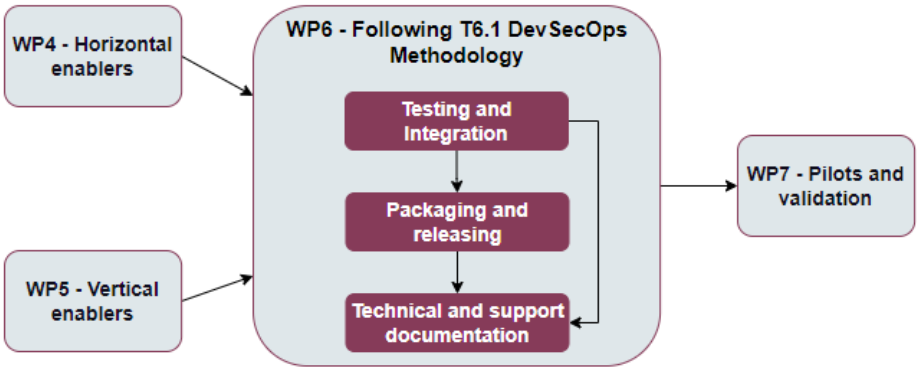
List of acronyms

Acronym	Explanation
API	Application Programming Interface
BSD	Berkeley Source Distribution (license)
CI/CD	Continuous Integration and Continuous Delivery/Deployment
CRD	Custom Resource Definition
DLT	Distributed Ledger Technology
GNU GPL	GNU General Public License (license)
GWEN	Gateway and Edge Node
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
K8s	Kubernetes
LTSE	Long-Term Storage Enabler
MIT	Massachusetts Institute of Technology (license)
MR	Mixed Reality
NGIoT	Next-Generation Internet of Things
OS	Operating System
OSM	Open Source Mano
PVC	Persistent Volume Claim
SAST	Static Application Security Testing
SCTP	Stream Control Transmission Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VPN	Virtual Private Network
YAML	YAML Ain't Markup Language

1. About this document

This deliverable corresponds to the second iteration of a series of two deliverables related to the packaging and releasing activities performed within WP6. The main objective of this document is to present all the actions carried out, updating the strategy and decisions presented in the previous iteration of the deliverable (D6.4, twelve months ago). These actions include the final planning, the scripts implemented in the CI/CD pipeline, the packaging strategy, conventions and tools considered/developed, the refined registries and licensing strategy for delivering the code and, although slightly out of the scope of the project, a brief summary of how artifacts can be then configured and deployed (i.e., operations phase of CI/CD). Because of the project extension, WP6 outcomes will be reviewed in a new scheduled deliverable, D6.8, including any refinement or change of the status of the enablers (in terms of packaging and releasing).

1.1. Deliverable context

Keywords	Lead Editor
Objectives	**O1:** This deliverable contributes to the implementation of an NGIoT architecture, by refining the release planning, licensing strategy and leveraged tools for packaging and storing the developed artifacts (in the form of enablers). **O2 to O5:** All the specific implementations associated to these objectives will be delivered as packages defined in this document. **O6:** The enablers releases will feed pilots, which will be the places where these will be validated.
Work plan	The task associated to this deliverable (T6.3) deals with all releasing aspects of the enablers developed in WP4 & WP5. The main outputs are the packaging structure and tools, followed with the CI/CD scripts to automate the process of packaging the enablers as standalone software artifacts and releasing them into project-level and external registries and repositories. 
Milestones	This deliverable directly contributes to MS7 – *Integrated solution* (initially planned for M30), as it sets the basis for the packaging of the software outputs of the project. In any case, a final review will be made on M36, in the final deliverable of the work package (D6.8).
Deliverables	This deliverable is the output of Task T6.3 – Packaging and Releasing. It is complemented with other concurrent deliverables of WP6, namely those devoted to integration and testing (D6.3) and the updated documentation (D6.6). It partially draws from D6.1 – DevSecOps methodology delivered in M6. The last deliverable of this WP (D6.8) will contain outcomes from all the tasks, although main focus will be on testing and integration rather than packaging, which basis and supporting CI/CD scripts are well-established.

1.2. The rationale behind the structure

Since the structure has been changed from the previous iteration, it has been deemed suitable to provide some explanation. D6.4's structure divided the sections following the DOA's T6.3 description, namely into: release strategy, release methodology, release cycle and release plan. However, it was hard to understand the differences and boundaries among these sections just by reading their titles. Hence, this version structures the content in a different way, in particular:

- Section 2 presents the **update of the release plan**, including the position of the packaging and releasing aspects in the DevSecOps approach.
- Section 3 presents the updated **outcomes related to packaging**, including the technologies considered, the conventions followed, the supporting tools implemented and the status of the enablers. Packaging is automated in the pipeline (see, Section 5), thanks to the files prepared per enabler following the guidelines presented in this section.
- Section 4 focuses on **releasing aspects** of the produced software artifacts, namely registries, repositories and licenses considered, involving code, containers and packages, within project execution and afterwards. Releasing is also automated in the pipeline (see, Section 5).
- Section 5 delves into **automation** of the packaging and releasing, depicting how they are integrated in the CI/CD pipeline and the scripts implemented (including security tests).
- Then, although a bit out of the initial scope of the deliverable, some guidelines for configuring and deploying an enabler are given in Section 6. Finally, conclusions are provided in Section 7.

1.3. Lessons learnt

Technically, many insights have been gained during the execution of the tasks associated to this deliverable:

- Enablers with many components require interactions (service names, ports, etc., pointing from ones to others). Supporting automated mechanisms are needed at packaging time to avoid manual configurations.
- When a chart of a third-party software is available, it is not needed to make a new one for it; still, some tailoring is needed to ensure a smooth integration with the ASSIST-IoT ecosystem.
- Although Helm is the *de facto* packaging standard for K8s, it does not provide support to Day-2 operations. Instead of fostering solutions like Juju that have this feature but hinders the development process (low community support), the use of operators and custom resource definitions are encouraged instead.

1.4. Deviation and corrective actions

As many development teams are involved, it is complicated that all follow the same CI/CD methodology and tests applied. Some groups have well-established methods to handle this, and the project does not aim at breaking their practices, but rather learn from them to design its own. Aiming at ensuring an adequate level of homogeneity among the developed artifacts, **packages (Helm charts) will follow a unified convention** (e.g., of structure of labels), **and will be released in common private and public registries** (as will happen with containers and released code). Overall, the project does not mandate to follow the exact pipeline developed but will ensure that results are coherent and integrated with the ASSIST-IoT environment, while allowing their deployment in other virtualised environments (i.e., not governed by the project orchestrator).

1.5. Version-specific notes

In comparison to the previous iteration, this deliverable (i) presents a more updated and concise report of the release planning, including the registries and licensing strategy; (ii) much curated guidelines for preparing (custom and existing) charts to be executed in an ASSIST-IoT environment, including the update of the Helm chart generator; and most importantly, (iii) presents the scripts leveraged to automate the processes of packaging (wrapping) and releasing into the involved registries and repositories.

2. Release and distribution plan update

The ASSIST-IoT project will release a large number of enablers (~40), providing several features related to horizontal planes or verticals of the reference architecture. Due to the specificities of the project, apart from *coding* the enabler components (or parts of them), the development phase requires to *build* them (into containers), *test* them (in a staging environment), *package* them (via preparing their Helm charts), and *release* them (into private and public repositories). The DevSecOps methodology (see D6.1 [ref]) has been followed for all the enablers, however, they have not been released at the same time since they widely vary in:

- Complexity. Some enablers are easier to realise than others (e.g., some require to be developed from scratch, while others require tailoring or composition from existing solutions – avoiding to reinvent the wheel).
- Criticality. Essential enablers have been considered more "urgent", aiming at having them earlier than the rest, as they will be part of (almost) all ASSIST-IoT implementations.
- Coupling level. Some enablers work in pure standalone fashion (e.g., self-resource provisioning enabler); others are part of a larger framework (e.g., Federated Learning enablers); and others will require integration at an "architecture" level, which require greater communication between different teams.

Hence, having a unified release plan for all the enablers of the project has not been suitable nor feasible. Two main "platform-level" release dates were set for the project: M18 (April 2022) and M30 (April 2023). In the former, a **functional version of the essential enablers** was released, namely: smart orchestrator, long-term storage enabler, edge data broker, VPN enabler, tactile dashboard, Open API, identity manager, authorisation enabler, DLT logging enabler, and manageability enablers. Not all these enablers were packaged at that time, as some partners were not that familiar with the packaging technology chosen (i.e., Helm). The second release date, concurrent to the present deliverable, reports the packaging and releasing status of **all the enablers**, which should have a functional version ready and **packaged** (encapsulation exceptions excluded, i.e., cybersecurity agent, self-healing enabler and MR enabler). The release plan can be seen in the following figure:

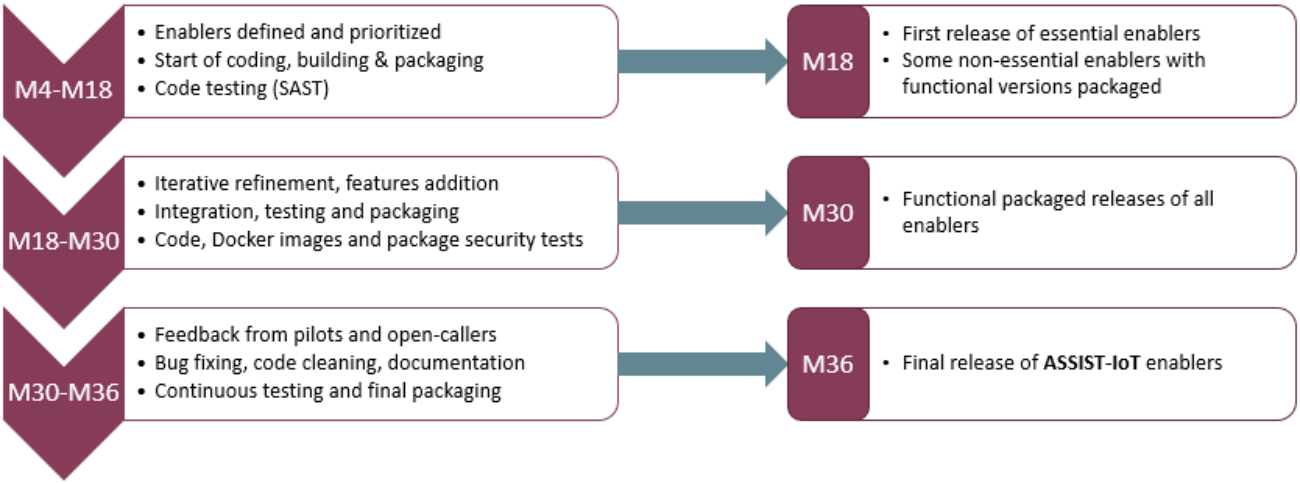


Figure 1. Release plan

It should be highlighted that, for this second release date, all enablers (including those from the first release) have been properly tested in a staging environment, as one can see in D6.3 (concurrent to this deliverable). Still, some actions are pending, including some integrations and the execution of end-to-end and acceptance tests (to be reported in M36 – October 2023), and hence a third phase will be executed. Regarding the second release:

1. It is accompanied by a set of guidelines and recommendations for setting-up the infrastructure topology and some software pre-requirements (see D6.6, released concurrently).
2. It includes a set of scripts to facilitate an instantiation of the system, including (i) the setup of the computing nodes (i.e., Kubernetes – kubeadm/k3s masters and workers, and plugins installation), and the particular (ii) top-tier node setup, which consists of the manageability enablers, the tactile dashboard and the smart orchestrator (see D6.6 annexes) to deploy and connect the rest of enablers.

3. It is complemented with an update of the ReadtheDocs, consisting of wikis with guidelines for configuration, deployment and usage of the enablers (link).

Developments are expected until this month (April 2023). However, since technical work packages have been extended by 6 months, some actions will be done to refine the overall solution: (i) implementing pending functionalities; (ii) fixing bugs found by open callers or in pilot deployments; (iii) improving or providing supporting material – automation scripts, video demos…; (iv) finishing documentation; and (v) cleaning the code structure of the enablers. Any bugs or errors found once enablers are deployed in any of the pilots will require on an update of them, which will come as patch or minor releases. These actions will entail further testing and packaging, apart from the pending actions reported above, and will be carried out during the third phase.

3. Packaging

As commented in the previous iteration of the deliverable (D6.4), ***Docker***[1] has been chosen as the main **virtualisation technology** because of its relative simplicity, wide adoption and ease of use. Still, Docker alone **lacks functionalities** that are essential **for production-ready environments**, so being the *de facto* standard, ***Kubernetes***[2] **(K8s)** was chosen as container orchestration framework to bring these features. Also, an enabler will require several K8s manifests to be deployed (at least two per component, workload and service related, but can be extended to PVCs, ConfigMaps, Secrets, CRDs, etc.), along with the underlying Docker images. Aiming at avoiding having a large number of separated s manifests with static configuration, ***Helm***[3] packaging technology was chosen to **wrap enablers** in a single file, and **ease configurations**: Helm is a "*package and deployment manager for K8s, working similarly to apt in Linux systems; the K8s resources required for an application are packaged in a chart. Helm manages the complete lifecycle of an application, (or rather, of a chart), from instantiation to termination. It also allows templating, that is, instead of having K8s manifests, charts contain templates, which include placeholders for the values of the specified K8s resources. When a chart is instantiated, actual manifests are generated, substituting these placeholders by real values, which usually come from a file which contains these values. This feature is very interesting as an application can be easily customised for a particular environment, without having to modifying the code from the templates*". Extended rationale of the selection of the aforementioned Cloud Native technologies and the concepts related to Helm can be seen in D6.4.

Charts (see Figure 3, D6.4) must be provisioned for each enabler to be packaged. Then, the wrapping of an enabler is performed by the pipeline (see Section 5.1.4) considering those files. ASSIST-IoT fosters its own, tailored (hybrid) strategy for preparing charts: an enabler might have components realised by custom, project-level code and third-party code. This means, that the chart will contain project's component templates, while third party charts will be added as dependencies (see Figure 2 below, right approach, also called "sub-charts"). An example of the latter could be an SQL database component (e.g., MariaDB chart) of an enabler, from which making a chart from scratch would not make sense nor add value. With the proposed, hybrid strategy, it is easier to maintain a mixture of project's and external code.

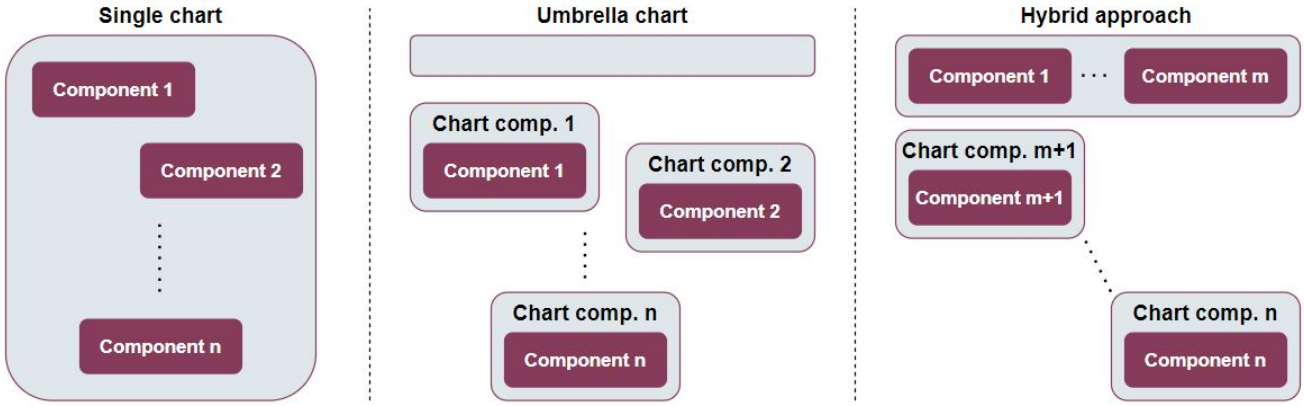


Figure 2. Helm chart structuring strategies

In any case, ASSIST-IoT guidelines do not mandate any particular structure; enablers will work regardless of the strategy considered, and DevOps cycles can be adapted to any of them (i.e., with dedicated scripts in the DevSecOps pipeline). Hereinafter are provided a set of minimum guidelines aiming at easing the preparation, structuring and packaging of an enabler's chart (considering existing ones, e.g., for dependencies, or preparing a new one for the project). It should be highlighted that guidelines and suggestions will not be exhaustive, as it is not the objective of this document to increase the packaging burden, but rather to provide a minimum structuring pattern for ASSIST-IoT enablers.

[1] https://www.docker.com/
[2] https://kubernetes.io/
[3] https://helm.sh/

Charts of ASSIST-IoT follow the official conventions and best practices posed by Helm. Hence, the official structure is kept, considering the typical folders (*charts* – for the dependencies; templates – for the components' files; and *crds* – for the custom resources, when implemented) and files (*Chart.yaml* – with the main information of the chart, included sub-charts info; *values.yaml* – with configuration/deployment options applied to the templates to realise manifests at deployment time; and the typical *LICENSE* and *README* files). In addition, this structure is extended with two minor modifications: first, by **including a folder structure within the templates folder**, to separate the templates related to each components developed (not from third-parties, those are kept separately in their respective charts, under the charts folder); and second, by including a *qa-values.yaml* for having configuration parameters for development phases (e.g., for the staging environment) separated to the official one. The structure proposed is presented in Figure 3.

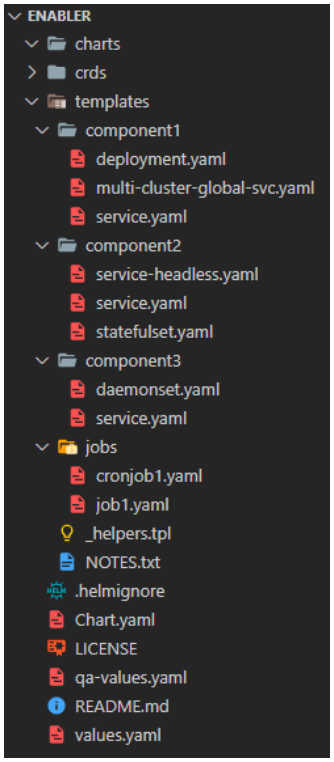


Figure 3. Proposed enablers' chart structure

The next subsections delve into the actions that developers should make to either adapt existing Charts (previously-developed or from third parties), or make new charts from scratch with the generator. **Notice that every guideline commented in Section 3.1 is included in the charts produced with the generator v3.0.0+.**

3.1. Adapt existing charts

Apart from the extended structure, different conventions and logical structures (in terms of *labels, annotations, helpers* in the templates, and in the *values.yaml*) have been defined to smooth later integration with the infrastructure, mainly with the networking and mesh plugin (i.e., Cilium) and the Smart orchestrator (mainly when it is commanded to decide the optimal place of the enablers within the managed continuum). To adapt enablers previously developed, or existing charts, to the ASSIST-IoT ecosystem, **a developer must**:

1. Add the ASSIST-IoT defined labels, which are based on the recommended labels[4] in the K8s official documentation, for all the components in the *_helpers.tpl* file. These labels are divided into

   (i) labels (those used to describe a component and also include the selector labels):

   - **helm.sh/chart**: a custom value composed by the chart name (defined in the *Chart.yaml*), a hyphen, and the chart version.
   - **app.kubernetes.io/version**: the appVersion of the *Chart.yaml*.

[4] https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/

- **app.kubernetes.io/managed-by**: the tool being used to manage the operation of an application, which will always be "Helm".

and (ii) selector labels (those used to select the proper component regarding K8s internal operations):

- **app.kubernetes.io/name**: the name of the enabler.
- **app.kubernetes.io/instance**: a unique value identifying the instance of an enabler to distinguish multiple deployments of the same enabler in a K8s cluster. When using Helm, it is recommended to use the release name used in the chart installation.
- **enabler**: the name of the chart defined in the *Chart.yaml*.
- **app.kubernetes.io/component**: component name.
- **isMainInterface**: a boolean value (a string with "yes" or "no" value) to define if the component is a main interface of the enabler.
- **tier**: defines the internal tier of the component inside the enabler, for instance, api, database, backend, frontend, …

Specifically, developers must create custom names and labels for each component and job **in *_helpers.tpl*** (remember to replace componentN name by the correct one/ones):

```
{{/*
Create chart name and version as used by the chart label.
*/}}
{{- define "enabler.chart" -}}
{{- printf "%s-%s" .Chart.Name .Chart.Version | replace "+" "_" | trunc 63 |
trimSuffix "-" }}
{{- end }}
```

```
{{/*
Component component1 labels.
*/}}
{{- define "component1.labels" -}}
helm.sh/chart: {{ include "enabler.chart" . }}
{{ include "component1.selectorLabels" . }}
{{- if .Chart.AppVersion }}
app.kubernetes.io/version: {{ .Chart.AppVersion | quote }}
{{- end }}
app.kubernetes.io/managed-by: {{ .Release.Service }}
{{- end }}
```

```
{{/*
Component component1 selector labels.
*/}}
{{- define "component1.selectorLabels" -}}
app.kubernetes.io/name: {{ include "enabler.name" . }}
app.kubernetes.io/instance: {{ .Release.Name }}
enabler: {{ .Chart.Name }}
app.kubernetes.io/component: component1
isMainInterface: "yes"
tier: {{ .Values.component1.tier }}
{{- end }}
```

And **add the labels in each manifest of each component and job**:

```
spec:
  selector:
    matchLabels:
      {{- include "component1.selectorLabels" . | nindent 6 }}
  template:
```

```
    metadata:
      labels:
        {{- include "component1.labels" . | nindent 8 }}
```

2. The Cilium multi-cluster global service feature allows to have the possibility of consuming an enabler deployed in the cloud (cluster with the smart orchestrator) by its name, from anywhere at the managed continuum (for instance, LTSE can be called by name from edge clusters, without introducing IP address or ports). To implement it, a developer needs to:

   (i) create a boolean trigger object in ***values.yaml***:

```
# Cilium Multi-cluster global service.
globalService: false
```

   (ii) create a **new manifest** for the MultiClusterService and include it in the enabler's chart:

```
{{- if .Values.globalService }}
apiVersion: assist.eu/v1
kind: MultiClusterService
metadata:
  name: {{ include "component1.fullname" . }}
  namespace: {{ .Release.Namespace | quote }}
  annotations:
    io.cilium/global-service: "true"
  labels:
    {{- include "component1.labels" . | nindent 4 }}
spec:
  ports:
    - name: main
      port: {{ .Values.component1.service.ports.main.port }}
      targetPort: {{ .Values.component1.service.ports.main.targetPort }}
      protocol: {{ .Values.component1.service.ports.main.protocol }}
{{- end }}
```

   (iii) create the corresponding annotations as a constant in ***_helpers.tpl***:

```
{{/*
Cilium Multi-cluster global service annotations.
*/}}
{{- define "globalServiceAnnotations" -}}
io.cilium/global-service: "true"
io.cilium/service-affinity: remote
{{- end }}
```

   (iv) include these annotations in all the **services** that are desired to be exposed as multi-cluster global services:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    {{- if .Values.globalService }}
    {{- include "globalServiceAnnotations" . | nindent 4 }}
    {{- end }}
```

3. To enable the feature of deploying all the components (and jobs) of the enabler **in a specific node** or in a specific selection of them, (i) create a boolean trigger object in *values.yaml*:

```
# Deploy all the components in specific K8s node(s).
enablerNodeSelector: {}
```

(ii) include this new nodeSelector in all the manifest of the enabler's components and jobs. In this example is defined a logical structure in which is used by default the enablerNodeSelector value, and if it is not present, the default nodeSelector value is used:

```
{{- with .Values.enablerNodeSelector }}
nodeSelector:
  {{- toYaml . | nindent 8 }}
{{- end }}
{{- if not .Values.enablerNodeSelector }}
  {{- with .Values.component1.nodeSelector }}
nodeSelector:
    {{- toYaml . | nindent 8 }}
  {{- end }}
{{- end }}
```

3.2. Helm chart Generator updates

This tool has been developed to help enablers developers prepare their charts, following the aforementioned structure and conventions. This tool is provided both as an executable and as Docker image, delivering a specific structure depending on the answers given by a developer, in a wizard-like fashion. The steps to use it are presented below:

3.2.1. Run in binary

Download the binary executable file that suits the machine OS (Linux, Windows and MacOS x64) and run it specifying the path to the folder where will be stored the generated chart. If the path is not specified, the generated chart will be stored inside a folder named "generated-charts" in the same location of the binary by default.

```
helm-chart-generator -o <path_in_machine>
```

3.2.2. Run in Docker

It's needed to create a simple volume to store the generated charts in the host machine and run the container in interactive mode. Different options are available:

1. Using the Docker image from Dockerhub:

```
docker run -it --rm --name helm-chart-generator -v <path_in_host_machine>:/chart-generator/generated-charts ravaga/assistiot-helm-chart-generator
```

2. Using the Docker image from the ASSIST-IoT's GitLab:

```
docker run -it --rm --name helm-chart-generator -v <path_in_host_machine>:/chart-generator/generated-charts gitlab.assist-iot.eu:5050/wp6/t6.3/helm-chart-genera-tor
```

3. Using the Docker image from the ASSIST-IoT's enabler public repository of GitLab:

```
docker run -it --rm --name helm-chart-generator -v <path_in_host_machine>:/chart-generator/generated-charts gitlab.assist-iot.eu:5050/enablers-registry/pub-lic/helm-chart-generator
```

3.2.3. Interaction with the wizard

Regardless of how it is run, a user of the generator must answer the **general questions**:

- **Enabler name**: the enabler name in lowercase and without symbols or spaces (only hyphens are allowed; capital letters, spaces, underscores, dots and slashes will be automatically removed). Try to not include the word "*enabler*".
- **Description**: description of the enabler or the chart.
- **Chart version**: version of the chart in Semantic Versioning (x.y.z).

- **App version**: version of the enabler in Semantic Versioning (x.y.z).
- **Number of components**: number of components (Deployments, StatefulSets and DaemonSets) of the enabler (without including Jobs and CronJobs). The minimum number of components is 1. A service is created for each component, and if the component type is StatefulSet, a headless Service is additionally created.
- **Number of Jobs**: number of K8s Jobs of the enabler.
- **Number of CronJobs**: number of K8s CronJobs of the enabler.
- **Number of dependencies**: number of dependencies (subcharts) of the enabler.

Answer the specific ***component*'s questions** inquired by the generator (the first group of questions is related to the main component):

- **Component name**: the component name in lowercase and without symbols or spaces (capital letters, spaces, hyphens, underscores, dots and slashes will be automatically removed). Try to not include the enabler name.
- **Component type**: the K8s controller type of the component (Deployment, StatefulSet or DaemonSet).
- **Component image repository**: the container image repository of the component (e.g. ravaga/assistiot-helm-chart-generator or gitlab.assist-iot.eu:5050/wp6/t6.3/helm-chart-generator).
- **Component image tag**: the container image tag of the component (e.g. 1.5.2, development or latest).
- **Number of ports of the component's service**: number of ports of the K8s Service of the component. The default and minimum value is 1.

Answer the specific **component *service*'s questions** inquired by the generator:

- **Port name**: the port name in lowercase and without symbols or spaces (capital letters, spaces, hyphens, underscores, dots and slashes will be automatically removed).
- **Port protocol**: the port protocol (TCP, UDP or SCTP).
- **Port number**: the port number (allowed values range from 0 to 65535).

Answer the specific ***Cron* and *CronJob*** (if included) **questions** inquired by the generator:

- **Job/CronJob name**: the Job/CronJob name in lowercase and without symbols or spaces (capital letters, spaces, hyphens, underscores, dots and slashes will be automatically removed). Try to not include the enabler name.
- **Job/CronJob image repository**: the container image repository of the Job/CronJob.
- **Job/CronJob image tag**: the container image tag of the Job/CronJob.

Answer the specific ***dependencies***' (if included) **questions** inquired by the generator:

- **Dependency name**: the dependency name in lowercase and without symbols or spaces (only hyphens are allowed; capital letters, spaces, underscores, dots and slashes will be automatically removed).
- **Dependency version**: version of the dependency chart in Semantic Versioning (x.y.z).
- **Dependency repository**: Helm chart repository name (the repository must have been previously added using the *helm repo add* command) started with an @ (e.g., @bitnami) or its URL (e.g., https://charts.bitnami.com/bitnami).

Finally, the Helm chart will be generated inside the *generated-charts* folder. If the generator has been run using Docker, the chart will be generated inside the specified Docker volume

3.2.4. Generated structure and next actions

The generator will return the following structure (detailed in comparison with the presented above):

enablername/	
Chart.yaml	A YAML file containing general information about the chart.

.helmignore	The .helmignore file is used to specify files to not include in the chart.
values.yaml	The default configuration values for this chart. These values are grouped by the component or job that they belong to, inside a diferent object.
qa-values.yaml	A copy of the *values.yaml* file for development purposes.
charts/	A directory containing any charts (subcharts) upon which this chart depends. The folder is initially empty, so the user must include inside it the needed subcharts, or manage them dynamically using the "helm dependency update" command.
crds/	Custom Resource Definitions (empty folder).
templates/	A directory of templates that, when combined with values, will generate valid Kubernetes manifest files.
NOTES.txt	A plain text file containing short usage notes. A default file is generated.
_helpers.tpl	A place to put template helpers that you can re-use throughout the chart. In this file are defined the enabler name, chart name, component name and labels, jobs name and labels.
componentN	A directory containing the componentN K8s workload (Deployment, StatefulSet or DaemonSet) manifest and its Services manifests. Additional manifests can be included (e.g., ConfigMaps, PVCs, Secrets) inside this folder **by the developer**. **Note:** a directory is created for each component.
jobs/	A directory containing all the Jobs and CronJobs K8s manifests (only is created if the enabler has Jobs or CronJobs).

The above steps are not enough to deploy a specific enabler. Some manual modifications are additionally needed, including:

- Check in *values.yaml* if the ports of the service of all the components are correct. A specific NodePort can be specified for each service.
- Set the resources for all the components in *values.yaml*. *Limits* are required for the Smart Orchestrator when using the automatic scheduling capabilities, whereas *requests* are required for the Resource provisioning enabler.
- Set the **environment variables** for all the components and jobs in *values.yaml*, and then include them in the manifest (yaml file) of these components.
- For the StatefulSet components, configure the persistence object in *values.yaml*, and then in the manifest (*statefulset.yaml* file) of the component. By default, it is only created one persisted volume (one entry under the “volumeMounts” of the principal container and a logical block at the end of the file) for each StatefulSet component, so to create new persisted volumes, replicate the same structure in both values and manifest.
- Include in *values.yaml* the desired configuration of a dependency chart under the proper object, which coincide with the dependency name.
- If dependency charts have been added, they must be included inside the charts folder, or the chart installation will fail. This can be done by manually adding the charts inside the folder or automatically using the *helm dependency update* command.
- Add in *_helpers.tpl* additional constants, logics or labels. For example, generate in this file the value of environment variables that needs more complex structure (e.g., the Java Options environment variable in an Apache Tomcat container, the complete URL to connect to a database or a variable composed by some values of *values.yaml*), and then use this value in the component’s manifest.
- Copy the final content of *values.yaml* into *qa-values.yaml*, or configure the *qa-values.yaml* file with custom values for specific development purposes.

In enablers composed by multiple components, it is usually needed an interaction among them (e.g., an API that needs to access to a database), which is achieved using some protocols that need the service name, the port number, and additionally other parameters. For that reason, it is strongly advised to automatically set these

values to avoid the manual configuration of them in the *values.yaml*, or not repeat the configuration parameter multiple times. It is recommended to follow the next steps:

- For setting the service name of another component in the environment variables section of the component's manifest that wants to access to it, use the automatically generated constant in *_helpers.tpl* named <*component-name*>*.fullname*.

```
value: {{ include "database.fullname" . | quote }}
```

- For setting the service port of another component in the environment variables section of the component's manifest that wants to access to it, use directly the port number previously configured in the service object of the other component (*.Values.*<*other-component*>*.service.*<*port-name*>*.port*).

```
value: {{ .Values.database.service.ports.postgresql.port | quote }}
```

- For setting other parameters configured via environment variables of another component in the environment variables section of the component's manifest that wants to access to it, use directly the previously configured environment variables and avoid creating new ones with the same content. This also applies for the secrets.

```
value: {{ .Values.api.envVars.schema | quote }}
```

It is a good practice to perform some testing of the generated chart. Later on, the pipeline implements some more exhaustive testing (with KubeLinter tool, see Section 5), but still it is a good practice to spot some potential issues and try to solve them in advance:

1. Examine a generated chart for possible issues:

```
helm lint generated-charts/chart-name
```

2. Render chart templates locally and display the output. None of the server-side testing of chart validity is done.

```
helm template generated-charts/chart-name --debug
```

3. Test the installation of a generated chart without actually installing it (e.g. to inspect that the values of the *values.yaml* file are included in the K8s manifests, the labels are properly created, ...):

```
helm install <release-name> generated-charts/chart-name --debug --dry-run
```

   For example:

```
helm install test generated-charts/chart-name --debug --dry-run
```

If a YAML file is failing to parse, but it is interesting to display the generated output, comment out the problematic line in the template and re-run the above command (step 3). The output of the commented line will be displayed as a comment within the rest of the template.

```
env:
  - name: EXAMPLE_ENV_VAR
#    some: problem section
#    value: {{ .Values.api.envVars.exampleEnvVar }}
```

3.3. Enablers' packaging status

This section provides a summary of the packaging status of the enablers, with a summary of the exceptions, lessons learnt and mitigation actions, when this has not been possible or it is still ongoing.

Table 1. Packaging status of ASSIST-IoT enablers

Task	Enabler/s	Status	Comments
T4.2	Smart orchestrator	Pending, partial chart prepared (script-based).	This enabler is based on OSM orchestrator, which is based on a different packaging method. Project will deliver identical features without OSM in following months to allow such packaging.

	SDN Controller	Packaged	-
	Auto-configurable network enabler	Pending	This enabler has not been yet packaged as the feasibility of virtualizing its monitoring module is under analysis. Once the decision is taken, the enabler packaging will be performed, consisting of all its virtualizable components.
	Traffic classification enabler	Packaged	-
	Multi-link enabler	Packaged	API packaged. Pending to move some host-level features to virtualised realm during the next months, due to challenges related to virtualised network interfaces, before final packaging.
	SD-WAN enabler	Packaged	-
	WAN acceleration enabler	Packaged	-
	VPN enabler	Packaged	-
T4.3	Semantic repository enabler	Packaged	-
	Semantic translation enabler	Packaged	-
	Semantic annotation enabler	Packaged	-
	Edge data broker	Packaged	-
	Long-term data storage	Packaged	-
T4.4	Tactile dashboard enabler	Packaged	-
	Business KPI reporting enabler	Packaged	-
	Performance and Usage Diagnosis enabler	Packaged	-
	OpenAPI management enabler	Packaged	-
	Video augmentation enabler	Packaged	-
	MR enabler	Exception	See Section 5.4.1, D3.7
T5.1	Self-healing device enabler	Exception	See Section 5.4.2, D3.7
	Resource provisioning enabler	Packaged	-
	Location processing enabler	Packaged	
	Monitoring and notifying enabler	Packaged	-
	Automated configuration enabler	Packaged	-
T5.2	FL Orchestrator	Packaged	-
	FL Training Collector	Packaged	-
	FL Repository	Packaged	-
	FL Local Operations	Packaged	-
T5.3	Authorisation enabler	Packaged	-
	Identity manager enabler	Packaged	-
	Cybersecurity monitoring enabler	Packaged	-
	Cybersecurity monitoring agent enabler	Exception	See Section 5.4.2, D3.7
T5.4	Logging and auditing enabler	Pending, partial chart	The deployment of Hyperledger fabric on k8s and generating charts has been delayed due to
	Data integrity verification enabler		

	Distributed broker enabler	prepared (K8s manifests)	the complexity involved in the process. In order to ensure successful deployment, a specific sequence must be followed, which includes dealing with the complexity of PVCs and complying with the labelling conventions of services. As a result, creating Helm charts for Hyperledger fabric has proven to be a challenging procedure.
	DLT-based FL enabler		
T5.5	Enablers manager	Packaged	-
	Composite services manager	Packaged	-
	Clusters and topology manager	Packaged	-

4. Registries and licenses

4.1. Artifact registries

Three types of registries are needed: for the code, the containers and the charts. As depicted in the DevSecOps methodology deliverable (see D6.1), **GitLab** is the main **code repository** for the project. The structure of the different artifacts of the project into groups, subgroups and projects is presented, which in summary mirrors the division into work packages, tasks and enablers. As reported in the previous iteration of this deliverable, code will be migrated to **GitHub's ASSIST-IoT organisation (>= M36)** once final releases are available, making them available for the rest of the development community (applied primarily to WP4 & WP5 developments, without interfering with partners' internal policies related to code sharing). This action is expected to be completed by M36 (in six months' time).

Regarding containers, a similar strategy will be followed. **GitLab container registry** is being used to host the Docker images associated with the components developed within the project, making use of public images as base images for the enablers of the project, or whenever directly consumed. If prepared properly, images can be updated automatically via dedicated pipelines in GitLab. Once the developments associated to the components of an enabler are finished, the images will be uploaded to **DockerHub (>= M36)**, under the umbrella of an ASSIST-IoT repository. This has been selected as it is the most used and extended place to share virtualised software artifacts with the rest of the community. Aiming at making results available as soon as possible, latest container images are being made available at a public repository from GitLab (>= M30).

Finally, charts will also need a dedicated registry. Again, **GitLab package registry** is being used to host the charts developed in the project. As with the aforementioned registry, they can be associated to a particular project, being easy to manage code, containers and charts from a single location within the tool. In this case, once enablers are ready, charts will be uploaded to **ArtifactHub (>= M36)**, under the umbrella of an ASSIST-IoT repository. Again, the selection responds to its common use in the community with respect to other alternatives, and as with containers, results are already being shared in a public repository from GitLab (>= M30). A summary is given in Figure 4.

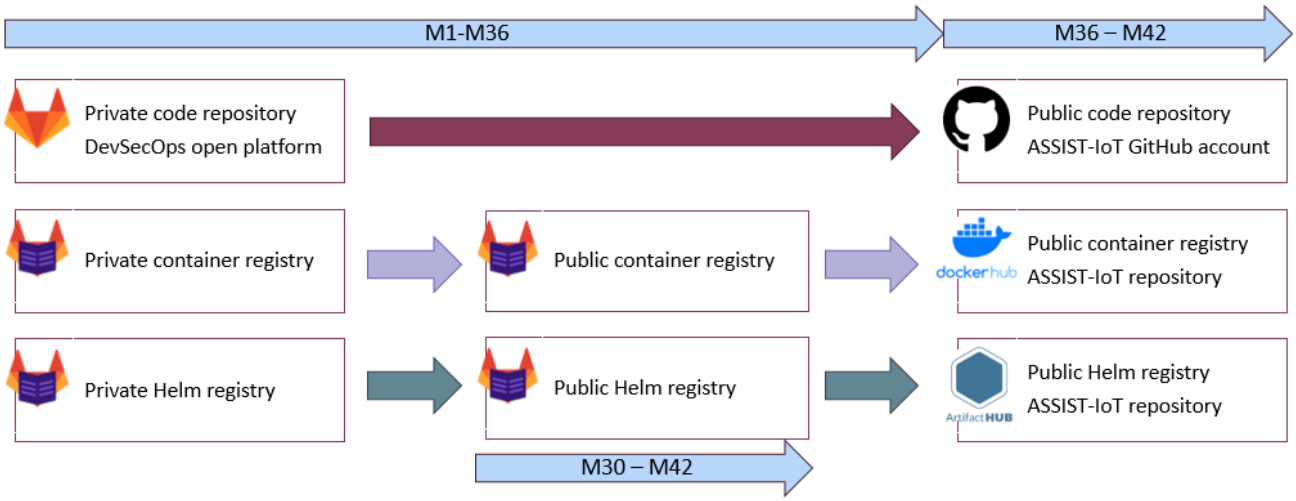


Figure 4. Artifacts registries

4.2. Licenses

A license defines the terms and conditions for using, reproducing, and distributing a product. In the case of software development, it defines the permission rights for utilising one or multiple instances of the software in ways where such use would otherwise potentially constitute copyright violation. There are several permissive open source software licenses with different terms, conditions and use cases. A summary of some of them is provided here. ASSIST-IoT will work on the basis of delivering its results considering **Apache license 2.0 or licenses with similar nature** (MIT, GNU GPL and BSD). This particular license is one of the most popular open-source licenses and belongs in the permissive category, allowing users to do anything they want with the

code, with very few exceptions. There are four requirements that have to be included by any user that makes use of software licenced under Apache 2.0: (i) the original copyright notice, (ii) a copy of the license itself, (iii) a statement if there are significant changes to the original code, and (iv) a copy of the NOTICE file with attribution notes. However, it is not necessary to release the modified code under Apache 2.0. Any simple modification notifications can be regarded as enough to comply with the license terms.

The use of any of these licenses will be accepted in order to distribute the results to the community. Kubernetes, one of the most popular open-source software options for container management, scaling and deployment is licensed under **Apache 2.0**, and as it will be used universally throughout the project, it guides the strategy for licensing. Nevertheless, partners reserve the right to apply different licensees and levels of code sharing in case of requiring additional protection (e.g., because of internal policies). Besides, all technical results from WP4 & WP5 will be make available for usage, even in the eventual case that code is not shared. Additional information will be provided in the final report associated to Task 9.4 (D9.7).

5. Relation with DevSecOps and CI/CD pipeline

In a typical DevOps cycle, a software is ready to be released once the software has been coded, built, integrated, tested, packaged and accepted. While testing and integration are scope of T6.2 (reported in D6.3), T6.3 (reported in the present deliverable) is in charge of the last steps of the pipeline (without considering operations). In this project, a DevSecOps methodology has been selected, entailing a reinforcement of security actions into the development lifecycle (automating security testing and integrating security into the CI/CD pipeline).

Security is often seen as a barrier to DevOps because it can slow down the development process if not addressed appropriately. Achieving DevSecOps harmony requires close collaboration between development, security (if present), and operations teams from the beginning of the development phase. The key to achieve this harmony is communication and collaboration between all teams, understanding their goals and objectives. By working together, they can each focus on their own area of expertise and ensure that the code is secure and can be deployed quickly and efficiently, working towards the same goal. Still, it is important to acknowledge that there is no one-size-fits-all, as the implementation of a DevSecOps strategy depends on the specific organisation and its needs.

In order to implement DevSecOps, **security testing tools need to be automated and integrated with the development tools and processes**. This will allow for security testing to be done continuously and automatically as new code is committed, resulting in the **integration of security testing as part of the CI/CD pipeline**. By doing this, security vulnerabilities can be identified and fixed early in the development process, allowing for security to be tested in the same way that the application is tested, before being deployed to production. It is important to have a security testing tool/s (there are many) that is/are easy to use and integrate with the development process. It is key that security-related tests are executed on a regular basis, ensuring that the application is continuously tested for security vulnerabilities as new ones appear over time. The reports generated by the security testing tool can be used to improve the overall security of an application. By following these steps, it is possible to get started with security testing and improve the security posture of the application, not just independently for each enabler but also afterwards for the platform as a whole.

In the packaging phase, GitLab SAST, Trivy and **KubeLinter** have been the **security testing tools** implemented by default in the pipeline, performing static security analysis of code, container images (and dependencies) and Helm charts, respectively. First, GitLab SAST will focus on code, "*scanning a variety of programming languages and frameworks, automatically running the right set of analysers even if the project uses more than one language*". Then, Trivy will target primarily container images, "*analysing dependencies, known vulnerabilities, misconfigurations, software licenses, sensitive information and secrets*". Finally, KubeLinter will "*evaluate Kubernetes manifests and Helm charts and checks them against a variety of best practices, focusing on production readiness and security. It will ensure that containers will run as a non-root user, enforcing least privilege, and that sensitive information is only stored in secrets.* The provided reports can help teams to check regularly security misconfigurations and DevOps-related best practices.

5.1. Scripts implemented

The pipeline has been divided into various stages, each of them with its own functionality and aiming at being as much automated as possible. In the following subsections, the different stages of the pipeline are commented, summarised in Figure 5. Two steps can be seen: **the merge request**, in which a developer requests merging a development branch into the main branch, and the **merge process** itself, in which the main code is updated. A repository maintainer/owner should analyse the reports from the request and the performance of an enabler in a staging environment before executing the merge, in which the updates from the development branch are

5.1.1. Lint of commit messages

Every new change that will be merged with the main branch of a GitLab repository has to be properly documented. To that end, appropriate semantic versioning of the code should be applied. The conventional commits "standard" is used in the pipeline to determine which type of version will be released every time a change in a development branch is done, and merge it with the main branch. To ensure the proper format in the

commit messages, this stage will be only applicable when a merge request to the main branch is created, as it will be mandatory for the merge request to have a valid commit message format.

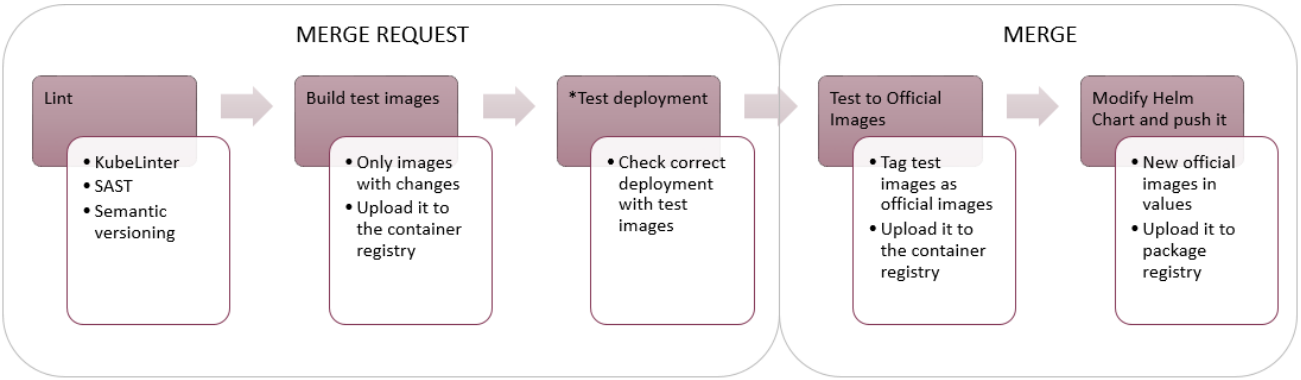


Figure 5. Packaging and releasing stages of the CI/CD pipeline

5.1.2. Lint Code

Following DevSecOps methodology, developer must ensure that the code that is uploaded and goes to production does not have dangerous vulnerabilities, dead or useless code. GitLab offers a template of SAST (Static Application Security Testing) that checks all the source code, highlighting the known vulnerabilities available on its database. This template is compatible with almost all the languages and frameworks in the market. It can be configured to only check for determined vulnerabilities or languages, if necessary. If there are warnings, a developer (e.g., a code maintainer) will need to review them and determine if they are critical or not; if accepted, the merge will be done.

Also, the pipeline integrates KubeLinter to check the K8s manifests and Helm charts submitted. Once the analysis is performed, a log with the warnings and errors is created, that could be read from the pipeline logs. The KubeLinter tool can be configured to check for specific rules, for example, to check for specific values in some manifests or to skip checks like non-root privileges. Like with GitLab's SAST tool, a reviewer must check the output with the vulnerabilities and determine if the merge request goes on.

5.1.3. Build and push Docker container images

Container images are the core of Cloud-Native applications, and thus also of ASSIST-IoT. The enablers' repositories should contain all the files needed to create their respective Docker container images, so they can be later uploaded to their respective container registries. However, as tokens are needed to access to each registry, having all enablers in separated ones can hinder their use. For this reason, there is a general repository with a common registry in GitLab, in which all the Docker images of the enablers will be also pushed after finishing this stage of the pipeline.

When a developer requests a code merge from a development branch to the main branch, the pipeline will only build the Docker images of the enabler components in which actual changes (in the code or the Dockerfile) have been submitted. Then, these images will be pushed with the tag "test", pending of validation. Building Docker images can get complicated when different platforms coexist, for instance, when using edge devices like Raspberry Pi's or the project GWEN in which enablers should comply with (e.g.,) arm architectures. This stage of the pipeline can be configured to build Docker images for other kind of architectures (e.g., x64, mostly used for typical cloud servers), although it is recommended that the developer ensures that the software developed is compatible with the architecture target.

Once the "test" Docker image of an enabler is ready and it has been validated by a repository maintainer/owner, the merge can be executed, and the images will be added to both the enabler and global container registries. There is no need to rebuild the image again; the pipeline takes care of downloading the image and substituting the tag "test" by the code of the last version.

5.1.4. Parse and push Helm Charts

Helm charts are composed of various files that describes the deployment characteristics. From the point of view of the CI/CD pipeline, this stage only will focus on the *values.yaml* file. Once a new Docker image is available, charts need to be modified to point to the new images built (i.e., names and tags). The pipeline takes care of doing so, by checking in the Container Registry the tag of the latest version of each image of the components belonging to the enabler. That makes it easier for the developer and they do not have to hardcode these data into the chart, ensuring the use the latest version of each image in the deployment.

Afterwards, the chart can be pushed into the Package Registry. As with containers, GitLab offers a package registry for every repo. To push the charts, the pipeline uses the same strategy that with Docker images: these are pushed in the enabler repo and in the general repo, tagging it with the last version the repo has at that moment.

6. Configuration and deployment options

The ASSIST-IoT architecture provides a Smart orchestrator and a set of interfaces (i.e., **manageability enablers**) to control the services' lifecycles, from their initial configuration to their deployment and termination. Under the curtains, Helm is supporting this process. Hence, an enabler (or group of them) can be implemented in any K8s-based system **interacting directly with Helm**, in order to deliver a specific functionality. Regardless of the case, specific configurations (day 0, day 1) are applied by modifying their default parameters in the *values.yaml* manifest, as all the environment variables of the underlying containers and several operational parameters (e.g., TCP/UDP port, autoscaling thresholds, update strategy, number of replicas, assigned hardware resources, volume locations, container image location, etc.) that should be modifiable by a user, are (or should be) specified in there. In the deployment phase, the default configuration can be modified in different ways:

a) With the Enablers' manager (from the manageability enablers), the field "Additional parameters" can be substituted by a JSON object containing the keys to modify (the object must follow the same schema as the values file), as seen in the figure below:

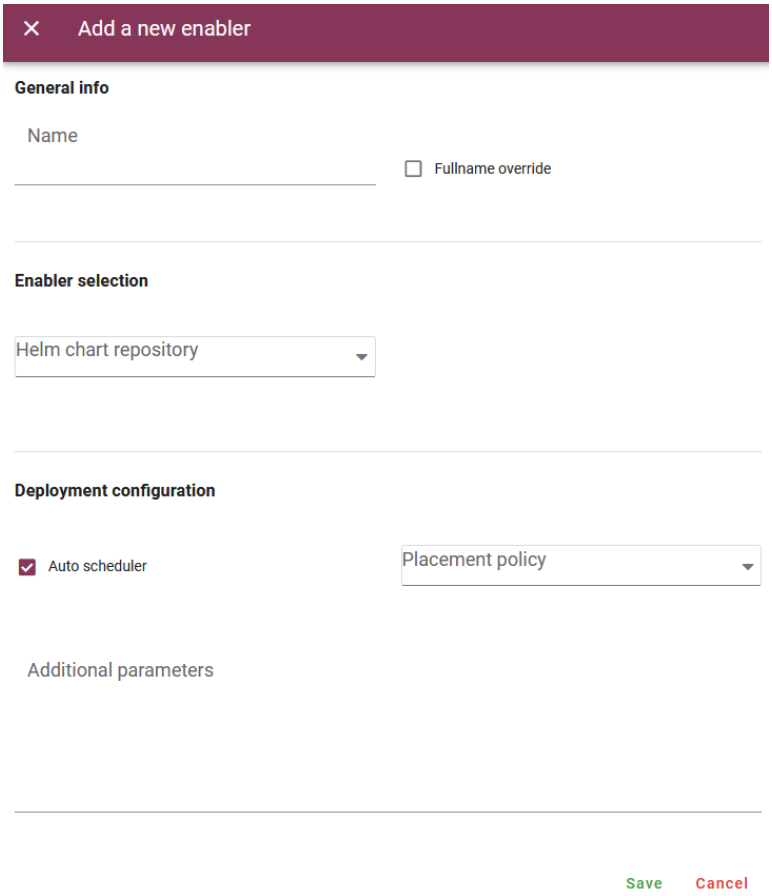


Figure 6. Configuring and deploying an enabler from the Enablers manager.

b) Interacting with the Smart orchestrator API directly: when deploying an enabler making use of the /api/enabler endpoint, the body of the HTTP call must contain the aforementioned JSON object in the "additionalParams" field:

```
{
    "enablerName":"enabler_name",
    "helmChart": "chart_repo/chart_name",
    "additionalParams": {    },
    "vim": "cluster_id",
    "auto": false,
    "placementPolicy":"worst-fit"
}
```

c) Interacting with Helm: when instantiating an enabler, replacing the values of one or many fields of the *values.yaml* manifest, via flag: helm **install**, with *set* flag pointing to the key and value to change.

d) Interacting with Helm: when instantiating an enabler, making use of an alternative file manifest (modified), via flag: helm **install**, with *values* flag pointing to an alternative manifest (which can modify all or some of the values of the manifest). In the project, the use of a *qa-values.yaml* also within the chart is considered, for configuration in the project's pre-production/staging environment.

Helm install is the command used for deploying a chart. Still, in the scope of the project, although it can be considered for testing/integration environments, it will not be the main mechanism for user operations (although it will be present underneath). ASSIST-IoT administrators will interact graphically via the manageability enablers, which act as intermediaries between users and the smart orchestrator API, facilitating the overall flow thanks to their graphical interfaces. Prior to deploying enablers, the user should have set up the clusters to manage and add Helm repositories. Additional information about its usage can be found both in D6.6 and in the project wiki[5]. The configuration and deployment flow is summarised in Figure 7.

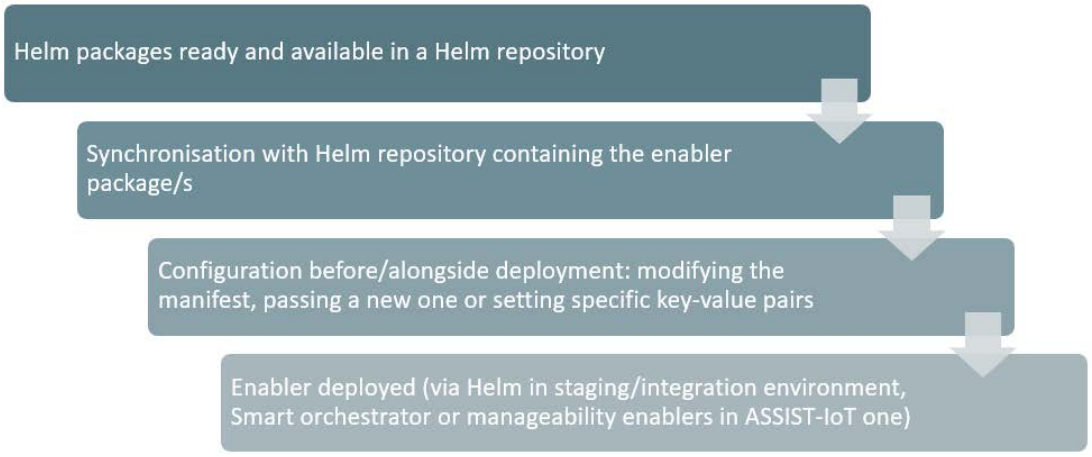


Figure 7. Configuration and deployment flow

[5] https://assist-iot-enablers-documentation.readthedocs.io/

7. Future work

This deliverable has reported all the actions carried out within the scope of T6.3 so far, focusing on packaging and releasing aspects: packaging strategy, associated tools, registries, licensing, etc., supported by automated tools in the CI/CD pipeline and incorporating security testing in the process.

Six month of task execution are remaining, and no major changes are expected in the structure of the presented framework. Still, different actions are envisioned:

- Refining the CI/CD pipeline, as more intensive use is envisioned for these months.
- Finalising the packaging of challenging enablers (i.e., Smart orchestrator, DLT enablers).
- Continue packaging and releasing the new versions of the enablers, as additional features and bug fixes will be implemented.
- Setting up the public registries of GitHub, DockerHub and ArtifactHub.
- Tackling the licensing of the developed artifacts, per enabler basis.

A. Enablers repository structure

A predefined structure of folders and files in each enabler repository is key for the efficient and organised management of the enablers. It helps users quickly find the necessary files, understand the relationship between them, and facilitate automated processes such as testing and deployment. With a clearly defined structure, scripts and pipelines can access and process files automatically, speeding up processes, and reducing the risk of human errors. This annex proposes a structure for the enablers repositories, based on best practices for folder structuring. This structure (see Figure 8) separates deployment from documentation and graphical material, considering the virtualisation and packaging technologies used in the project for deployment purposes (Docker, Helm charts), making it easier to locate files specific to certain encapsulation methods and preventing conflicts and duplications of information.

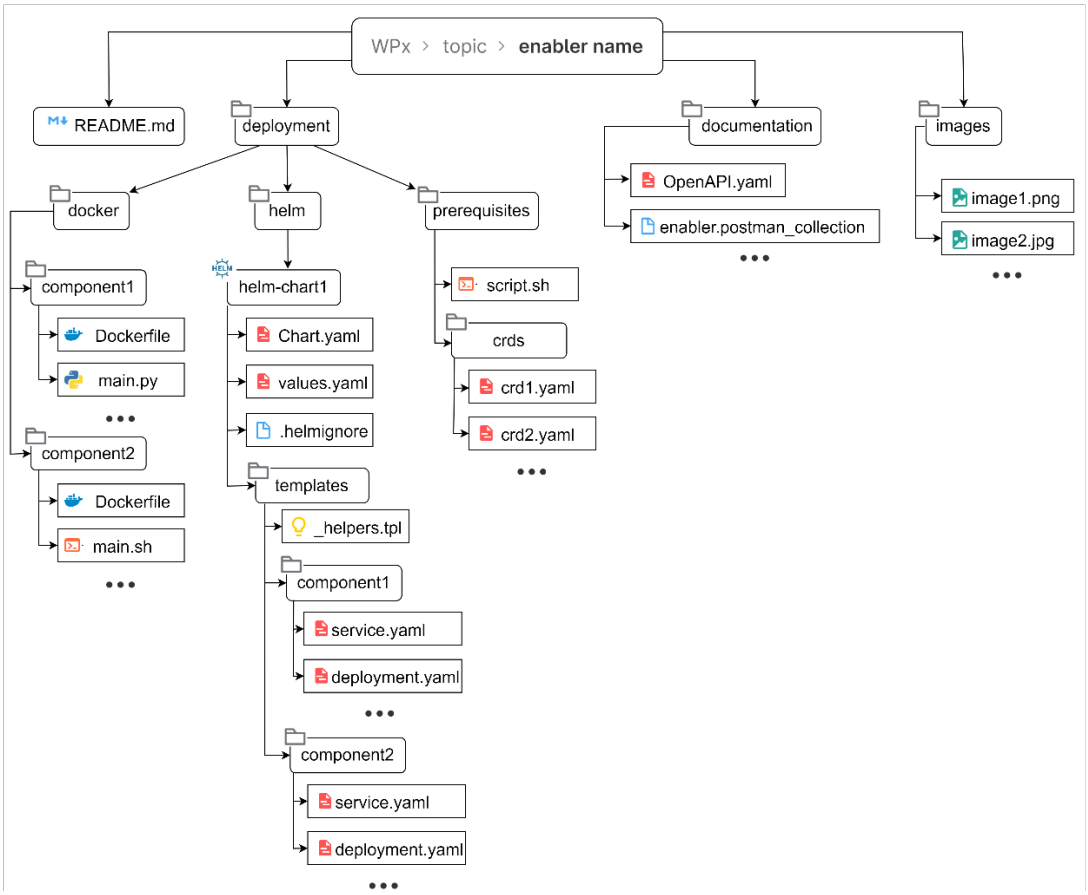


Figure 8. Enablers folder structure

README.md

The **README.md** file is essential in the main folder of a GitLab project, as it provides an overview of the project, its purpose, and how to use it. Additionally, it is an important part of the project's documentation process and helps to reduce repetitive user questions. For each enabler, this file will contain summarised information from the details collected in the ReadtheDocs.

Deployment

The **deployment** folder of the enabler repository will be the place where all the files related to the deployment of each enabler will be stored. To keep the deployment folder organised and manageable, it has been divided into subfolders based on the type of encapsulation used and the specific use case. For example, if an enabler has custom Docker images, it will need a folder with the files needed to create the custom image, this includes the "Dockerfile" and all other deployment scripts or files. The purpose of this organisation is to make it easier for users to find the specific deployment files they need and to avoid cluttering the deployment folder with irrelevant files. This also helps prevent any confusion or conflict between different deployment methods and ensures that all files are properly encapsulated.

By maintaining a clear and organised deployment folder, it will be easier to deploy and manage enablers in an agile and efficient manner. This can help reduce errors and improve overall performance, as well as make the deployment process easier to maintain and scale over time. The deployment folder is composed of the following sub-folders:

- **Docker:** It contains all the necessary information to create the required image(s) for each component, separated by folders with the name of each component. Within each folder, there will be a Dockerfile along with the required configuration and files. This step is important and necessary when using components that require custom images.
- **Helm:** It is one of the most important folders in the project repository, since it contains the configuration files for the Helm package management that will be used for the deployment of the enabler. This folder is organised in subfolders, one per component, containing the corresponding Helm Chart (with all its included templates/manifests, see Section 3. By default, each enabler will have a single Helm chart that has the same name as the enabler folder. However, there may be exceptions where an enabler requires multiple charts, depending on the complexity of the enabler and the different components that are part of it. Hence, it is important that the charts are properly distributed and documented in the "helm" folder. In addition, any file needed to deploy the enabler will be included, such as configuration files, Kubernetes manifests, installation scripts, YAML files and any other relevant files. Subfolders can also be included to further structure the files, if necessary.
- **Prerequisites:** It contains all the necessary configuration files required before installing an enabler, necessary for its setup and configuration in a particular deployment as smooth and error-free as possible. These files can be related to, e.g., creating needed namespaces, configuring network policies or setting up storage classes, and they can be in the form of bash scripts, K8s manifests, Helm charts, or any other relevant format. Proper organisation and clear documentation of these files is crucial for easing their use, as they will be usually applied manually by a user/administrator.

Documentation

This folder will contain all files related to the enabler's documentation, such as usage instructions, endpoint information, user guide, API call collections, and most importantly, an **OpenAPI** file. This file can be in YAML or JSON format and will contain the schemas and URLs of the endpoints to generate functional tests. This file must also be accessible through the corresponding call to the common endpoint "/api-export".

Images

The **images** folder will contain all images related to the enabler that can be included in the README.md or other files as example.