*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

# D6.4 Release and Distribution Plan – Initial

| **Deliverable No.** | D6.4 | **Due Date** | 30-Apr-2022 |
|---|---|---|---|
| **Type** | Report | **Dissemination Level** | Public |
| **Version** | 1.0 | **WP** | WP6 |
| **Description** | Software and documentation release and distribution plan to be followed for all components. First release will provide the plan, while second version plan will present the final execution outcomes. | | |

# Copyright

Copyright © 2020 the ASSIST-IoT Consortium. All rights reserved.

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

This document contains material, which is the copyright of certain ASSIST-IoT consortium parties, and may not be reproduced or copied without permission. This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

The information contained in this document is the proprietary confidential information of the ASSIST-IoT Consortium (including the Commission Services) and may not be disclosed except in accordance with the Consortium Agreement. The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the Project Consortium as a whole nor a certain party of the Consortium warrant that the information contained in this document is capable of use, nor that use of the information is free from risk, and accepts no liability for loss or damage suffered by any person using this information.

The information in this document is subject to change without notice.

The content of this report reflects only the authors' view. The Directorate-General for Communications Networks, Content and Technology, Resources and Support, Administration and Finance (DG-CONNECT) is not responsible for any use that may be made of the information it contains.

# Authors

| Name | Partner | e-mail |
|---|---|---|
| Alejandro Fornés | P01 UPV | alforlea@upv.es |
| Rafael Vañó | P01 UPV | ravagar2@upv.es |
| Ignacio Lacalle | P01 UPV | iglaub@upv.es |
| Evripidis Tzionas | P04 CERTH | tzionasev@iti.gr |
| Angeliki Papaioannou | P06 INF | apapaioannou@infolysis.gr |
| Nick Vrionis | P06 INF | nvrionis@infolysis.gr |
| Óscar López | P13 S21Sec | olopez@s21sec.com |
| Jordi Blasi | P13 S21Sec | jblasi@s21sec.com |
| Josue Moret | P13 S21Sec | jmoret@s21sec.com |

# History

| Date | Version | Change |
|---|---|---|
| 22-Dec-2021 | 0.1 | ToC closed and assignment of sections to partners |
| 25-Feb-2022 | 0.2 | First round of contributions |
| 21-Mar-2022 | 0.3 | Second round of contributions |
| 6-Apr-2022 | 0.4 | Third round of contributions |
| 7-Apr-2022 | 0.9 | Integrated version to IR |
| 30-Apr-2022 | 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 – Alex van den Heuvel, P02 SRIPAS – Katarzyna Wasielewska-Michniewska |

# Executive Summary

This deliverable is written in the framework of WP6 – Testing, Integration and Support of **ASSIST-IoT** project under Grant Agreement No. 957258. The document is the first of a series that are devoted to release and distribution of the software artifacts (i.e., enablers) of ASSIST-IoT project. In particular, this first version focuses on methodology, technological decisions, cycle and overall planning related to packaging and release aspects of DevSecOps.

With the aim at contextualising, the release strategy is presented as a subset of the DevSecOps methodology provided by the project. The document is focused primarily on delivering a methodology and a set of guidelines for preparing the structure of the enablers prior to their packaging. It also delves into the selection of the main technologies involved in the packaging and releasing, namely: Helm for packaging (once container images have been built) and deploying, GitLab not just as private code repository but also as a registry for containers and Helm packages (this is, internal for the project), and finally Dockerhub and Artifacthub for hosting the public releases of ASSIST-IoT software containers and Helm packages, respectively. The licensing strategy is also addressed.

The release cycle is also presented. This includes how the packaging, the configuration and the deployment of enablers are executed, considering Helm, the smart orchestrator and the manageability enablers as deployment tools. The two latter will be the main configuration and deployment tools for ASSIST-IoT users, still, making use of Helm underneath. Finally, the release plan for the enablers developed in the project is presented, which considers two main dates: M18, in which the enablers selected as essential (see D3.6) must have a first released version, and M27, in which all enablers must have a functional version ready for deployment (non-essential enablers may have the first release in-between).

# Table of contents

# List of figures

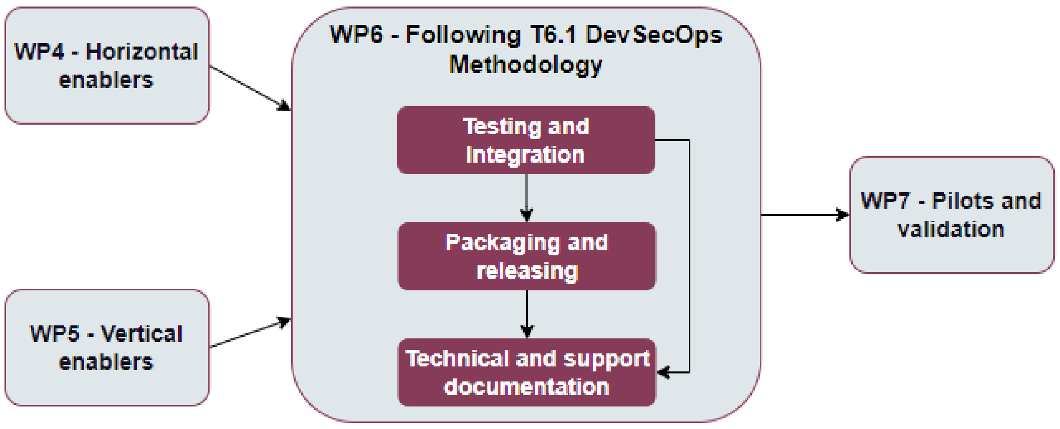
# List of acronyms

| Acronym | Explanation |
|---|---|
| **API** | Application Programming Interface |
| **CI/CD** | Continuous Integration/Continuous Delivery |
| **DevSecOps** | Development, Security and Operations |
| **DLT** | Distributed Ledger Technology |
| **HTTP** | HyperText Transfer Protocol |
| **IP** | Internet Protocol |
| **JSON** | JavaScript Object Notation |
| **K8s** | Kubernetes |
| **MIT** (license) | Massachusetts Institute of Technology |
| **SDK** | Software Development Kit |
| **SQL** | Structured Query Language |
| **URL** | Uniform Resource Locators |
| **VPN** | Virtual Private Network |
| **YAML** | Yet Another Markup Language |

# 1. About this document

The main objective of this document is to present the release methodology and planning that enablers developed within the scope of ASSIST-IoT will follow. The methodology and the technologies leveraged for the packaging and releasing of the enablers will be common, aiming at serving as guidelines and best practices rather than rigid and unchangeable mandates. The final refinements and outcomes as well as the delivered releases will be documented in the second iteration of the present deliverable, to be out in M30.

## 1.1. Deliverable context

| Item | Description |
|---|---|
| **Objectives** | **O1:** This deliverable contributes to the implementation of an NG-IoT architecture, by specifying the release methodology, cycle, licensing strategy and associated tools for the developed artifacts (in the form of enablers), alongside a high-level planning.<br>**O2 to O5:** All the specific implementations associated to these objectives will be delivered as packages defined in this document.<br>**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 aspects of delivery of the enablers developed in WP4 & WP5. The main output is the release and distribution plan, defining the packaging of compiled binaries and distribution of enablers as standalone software artifacts and virtualised components (containers and k8s manifests). Released enablers will be validated in the pilots of the action.<br> WP4 - Horizontal enablers<br>WP5 - Vertical enablers<br>WP6 - Following T6.1 DevSecOps Methodology<br>Testing and Integration<br>Packaging and releasing<br>Technical and support documentation<br>WP7 - Pilots and validation |
| **Milestones** | This deliverable does not mark any specific milestone completion. However, it contributes to MS7 – *Integrated solution* on M30, as it sets the basis for the packaging of the software outputs of the project. |
| **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 planning (D6.2) and documentation (D6.5). The next iteration is expected in M30. It partially draws from D6.1 – DevSecOps methodology delivered in M6. |

## 1.2. The rationale behind the structure

This deliverable follows a straightforward approach: in Section 2, the place of packaging and releasing within the DevSecOps cycle is introduced; Section 3 addresses common aspects related to packaging, hosting and licensing of the (packaged) software artifacts, focusing on technological choices and common strategies; in

Section 4 the cycle of packaging, configuration and deployment with the technologies to be considered in the project are described, along with the involved commands; a release plan for the enablers of the project is depicted in Section 5, differentiating the release of essential from non-essential enablers. Finally, conclusions are drawn in Section 6, explaining the outcomes expected in the next iteration of the current deliverable.

# 2. Release Strategy

## 2.1. Overview

In a typical DevOps cycle, a release is ready once the software has been coded, built, integrated, tested, packaged and accepted, being a key part of the delivery process (related with the CI/CD concept - Continuous Integration and Continuous Delivery/Deployment), as it can be observed in Figure 1. In this document, the release strategy entails also the packaging aspects, hence not covering (or barely) aspects related to the processes of building and integration, which are managed in a separated task and documented in a parallel deliverable (D6.2).

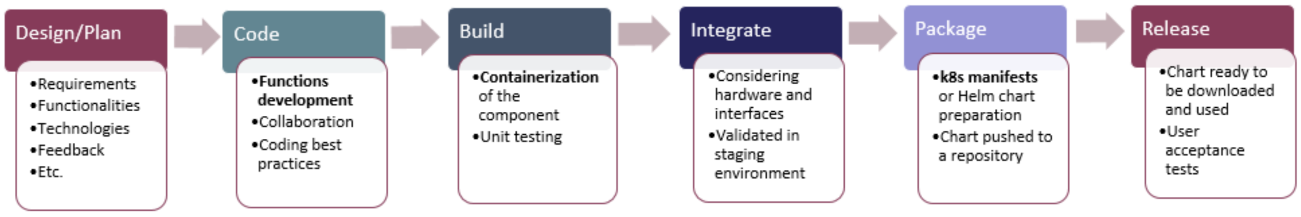

Design/Plan
•Requirements
•Functionalities
•Technologies
•Feedback
•Etc.
Code
•Functions development
•Collaboration
•Coding best practices
Build
•Containerization of the component
•Unit testing
Integrate
•Considering hardware and interfaces
•Validated in staging environment
Package
•k8s manifests or Helm chart preparation
•Chart pushed to a repository
Release
•Chart ready to be downloaded and used
•User acceptance tests

*Figure 1. Software development phases, from design to release*

## 2.2. Harmonisation with DevSecOps

DevSecOps is the process of integrating security into the software development lifecycle. This includes automating security testing and integrating security into the Continuous Integration/Continuous Delivery (CI/CD) pipeline to bring security to the process.

As a precursor of DevSecOps, DevOps conforms a set of best practices that combines software development (Dev) and IT operations (Ops), responding to the interdependence of the teams in charge of them. It aims to help organizations (i) shorten the software development lifecycle reliably, enabling the implementation of agile principles and practices more effectively, and (ii) having a better task (and software) organisation. Security is often seen as a barrier to DevOps because it can slow down the development process if not addressed appropriately. However, DevOps and security can work together to create a more secure and efficient software development lifecycle, integrating security into the various stages of the software development lifecycle.

Achieving DevSecOps harmony requires close collaboration between development, security, and operations teams. development teams need to be able to work quickly and efficiently, while still ensuring that the code they write is secure. Security teams need to be able to review code and identify potential security issues quickly and easily. And operations teams need to be able to deploy code quickly and securely. The key to achieving DevSecOps harmony is communication and collaboration between all three teams, understanding the goals and objectives of each team. 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.

By following a DevSecOps approach, organizations can create a culture of shared responsibility for security, which can help to identify and reduce the overall risk of security vulnerabilities in software applications. As described in D6.1 – DevSecOps Methodology, there are several challenges that organizations face when harmonizing DevSecOps, including:

- Ensuring that security is built into the development process from the start, rather than being an afterthought.
- Incorporating security testing and verification at every stage of the development process.
- Getting developers on board with the importance of security and making them aware of potential risks.
- Coordinating the work of different teams (development, security, operations) to ensure a smooth and secure development process.

DevSecOps is relatively new and thus is still evolving. Some key principles that are essential to creating a harmonious DevSecOps environment include communication and collaboration between development and operations teams (and security, if present), automation of security processes, and integration of security tools and technologies into the DevOps pipeline. It is important to acknowledge that there is no one-size-fits-all, as the harmony of DevSecOps depends on the specific organization and its needs. However, some common elements of a harmonious DevSecOps environment include a culture of collaboration, a focus on automation, and a commitment to continuous improvement.

In the process of getting DevSecOps deployed and automated, it is important to include the development team, operations team and security team from the beginning. If security is only an afterthought, it will be difficult to successfully deploy DevSecOps. It is also imperative to establish a process so that security is not an obstacle to the development and release of new features. By integrating security team to work with the other teams, it is possible to address security concerns and improve the security posture of the application. In order to achieve this, **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.

To achieve an effective security integration into the CI/CD pipeline, it is important to have a security testing tool/s that is/are easy to use and integrate with the development process. There are a number of tools available that can be used to automate security testing, and to test an application for a variety of security vulnerabilities. The integration of security tools within the pipeline will allow for security testing to be done automatically as new code is committed. It is important 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: an application considered secure in a specific date might not be some weeks or month later. 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.

The following sections present the **methodology, tools, cycles and plan related to release and packaging stages** that are being integrated in the ASSIST-IoT workflow. The involved tools will be integrated in the CI/CD pipeline, also considering the security tools of the project.

# 3. Release Methodology

This section introduces the main technologies, guidelines, repositories, and licensing strategies that will be followed to prepare the enablers for their further consumption, hence conforming the release methodology for the project. The rationale behind each decision is provided, and the main concepts related to them are introduced and summarised. This methodology will be complemented in the following sections with a summary of a cycle of execution (depicting the particular commands that will be involved from the beginning of a packaging until its deployment) and with an actual release plan for the enablers of the project.

## 3.1. Packaging

### 3.1.1. Technologies

Without considering architectural exceptions, enablers are designed following virtualization principles, in which their respective internal components are realized as containers[1]. Containers emerged to facilitate the deployment of applications in different computing environments, being a lightweight alternative to virtual machines. **In the first place, building entails preparing the container images of the components of the enablers**, in the DevSecOps flow (which should be realised considering automatic scripts, to convert a codebase into a container image). **Building can be understood as the first step towards a packaged solution** (in this case, enabler). In the scope of the project, ***Docker*[2] will be the main virtualisation technology** because of its relative simplicity, wide adoption and ease of use (alternatives such as *containerd*[3] images + command-line clients like *ctr*[4] or *nerdctl*[5] for instantiating containers is also feasible, but not as common). Docker includes an engine for running containers on top of a host Operating System, a command line interface to interact with the engine and the containers, a descriptor file known as Dockerfile, and additional tools, like Docker Compose.

Still, **Docker alone lacks functionalities that are essential in production-ready environments**. To begin with, Docker works in a single node, which is a clear point of failure: in case this node fails, all the services/ applications from the hosted containers are halted. Also, Docker alone does not support rollout nor rollback strategies (e.g., Canary or Blue-Green releases), meaning that if a container has to be updated because the application/service it contains has a new version, it must stop the current container instance and launch a new one, causing a service discontinuity. To solve this, Docker requires of a container orchestrator platform that brings additional capabilities for production-ready ecosystems. Among the existing options, Kubernetes[6] (abbreviated as k8s) is the *de facto* standard for production environments, overcoming by far alternatives such as Apache Mesos[7] or Docker Swarm[8]. Apart from supporting clustering of nodes and rollback/rollouts without service discontinuity, it brings other key functionalities such as scaling, load balancing, self-healing, observability and automatic (yet configurable) security mechanisms, extending the features provided by Docker.

Assembling an **application in k8s for a production environment requires preparing a set of manifests**, i.e., specifications of k8s API resources in JSON or YAML format. **This will be the main purpose of the packaging stage in ASSIST-IoT**. Some common resource manifests include[9]: (i) workload manifest (usually one of the following: Deployment, StatefulSet, DemonSet, Job or Cronjob), in charge of declaring the needs of the pods (minimum computation units that can be managed by k8s) and their replicas, which host the containers for a specific application; (ii) services, to give a unified entrypoint for all the replicas of a specific workload resource; (iii) persistent volume claims, to demand a physical resource to store data of the workload; and (iv) ConfigMaps and/or Secrets to store configuration or sensitive data in k8s, that can be later on passed to pods. On the one hand, not all the former are needed for every application; for example, a stateless application will not require

[1] Internal components of an enabler might not always be realised as an individual container. In some cases, for better use of resources, it is better to encapsulate them together although logically described as separated entities.
[2] https://www.docker.com/
[3] https://containerd.io/
[4] https://github.com/projectatomic/containerd/blob/master/docs/cli.md
[5] https://github.com/containerd/nerdctl
[6] https://kubernetes.io/
[7] https://mesos.apache.org/
[8] https://docs.docker.com/engine/swarm/
[9] https://kubernetes.io/docs/concepts/

persistent storage. On the other hand, other applications may require other manifests such as Ingress, ServiceAccount, NetworkPolicies, etc.

As mentioned before, **ASSIST-IoT enablers (likely) consist of more than one component**; each of them, containerized (individually or grouping some functionalities), requires a set of k8s manifests to be deployable in a production-ready k8s (multi-)cluster environment. Hence, for a single enabler, **several manifests may be in place**, which is not very practical neither to share nor to deploy. For this reason, a packaging tool and a structuring methodology is needed. This aspect can be addressed considering different “package managers” for Kubernetes, including:

- **Helm[10]:** It 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 customized for a particular environment, without having to modifying the code from the templates.
- **Kustomize[11]**: Unlike Helm, which defines a structure for the charts and encourages developers to substitute manifests by templates, with placeholders, Kustomize works directly over actual manifests. A developer should have all the manifests related to an application in a folder, and in case modifications or configurations are required for a particular environment or DevOps stage (development, production, etc.), variants can be included using *overlays*, which “patch” the predefined values of the manifests. Unlike Helm, Kustomize is managed by the k8s team, so it works alongside k8s instead of being installed as a separated tool.
- **Juju[12]:** It follows a different approach than the two previous tools. Templates and overlays are straightforward, quite useful for modifying the initial configuration parameters of a specific application. However, for real-time configurations where k8s resources are involved (Day-2 operations like backing up databases), they fall short. Juju provides an SDK to write *operators*, so once deployed, and while the application is running, users have the ability of reconfiguring parameters. In this case, applications alongside its operators are packaged in a *charm*. Another advantage with respect of the former tools is that, in case that two applications declare the same dependency with an external application, Juju will not launch two separated instances of it, but will connect the two former to a common, single instance of the latter (with Helm or Kustomize, some manual configuration must be made).

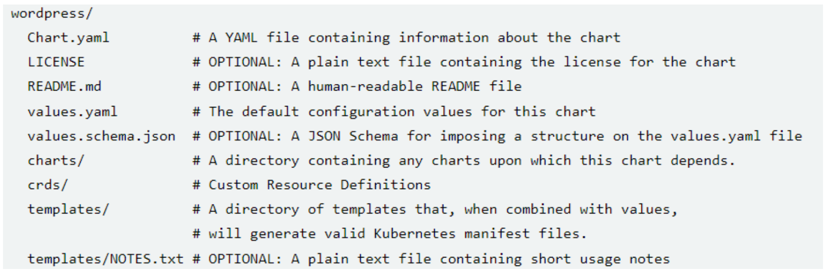
**Helm** was selected as the **packaging technology for ASSIST-IoT**, and the reasons behind this selection are listed as follows. First of all, Helm is currently the *de facto* standard, and it does not seem that this is about to change in the short term, which translates to the fact that many software solutions already have production-ready and maintained charts that can be easily adapted and used for the project. Second, it is about simplicity. From a developer’s perspective, although not complicated, preparing charts is not as fast as declaring kustomize overlays; however, from a user perspective, not expert, it is much simpler to modify a file with values (even easier if guided by a graphical user’s interface, as intended in the project). As a final remark, being compressed in a single file, charts are very portable and easy to register. In comparison to Kustomize, Helm’s scope is larger and supports more and wider functionalities, including hooks, rollouts/rollbacks and distribution server, being a “full” package manager. Besides, Juju shows very interesting functionalities, however, it has lower support and requires a very deep understanding of the application, Kubernetes infrastructure and dependencies in order to take full advantage of it, meaning that enablers’ development time might have greatly increased in case of devoting time to the development of dedicated operators. To conclude, once components are built, **packaging stage in ASSIST-IoT** will be in charge of **preparing the required k8s resources manifests (or rather, templates), wrapping them in a compressed file and upload the latter to a software artifact repository**, as automated as possible (i.e., with dedicated scripts in the DevSecOps pipeline). For facilitating the enabler packaging process, guidelines and structuring templates are provided in the Section 3.1.3.

[10] https://helm.sh/
[11] https://kustomize.io/
[12] https://jaas.ai/

### 3.1.2. Helm concepts

Some concepts should be introduced in order to become acquainted with Helm, especially as it will be the main packaging technology:

- Chart: Helm's packaging format. It is a collection of files that contains all the resource definitions to run an application, tool, or service in a k8s cluster. In Figure 2, an example of chart structuring of a WordPress application is depicted. Some key files and folders are the following:
    - Chart.yaml: It includes general information about the application and chart, including versioning and indicates software dependencies (also referred to as sub-charts).
    - values.yaml: It specifies the default values of the templates, which are used to build the k8s manifests in case these are not overridden. It can be used to modify values of the sub-charts.
    - charts folder: Hosts the actual chart dependencies, which are instantiated before the main chart.
    - templates folder: Hosts the template manifests, which will be later on rendered into actual manifests when the chart is instantiated.
- Repository: HTTP server where charts can be collected and shared. Helm provides an external tool named *Chartmuseum*, which can be used either as a simple local repository for development stages or integrated in commercial cloud storage backends.
- Release: Instance of a chart running in a k8s environment. In order to avoid confusing Helm's release concept with the homonymous concept in the DevSecOps methodology, "chart instance" will be used instead (Helm's release concept would match Deploy in a typical DevOps cycle).



```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE             # OPTIONAL: A plain text file containing the license for the chart
  README.md           # OPTIONAL: A human-readable README file
  values.yaml         # The default configuration values for this chart
  values.schema.json  # OPTIONAL: A JSON Schema for imposing a structure on the values.yaml file
  charts/             # A directory containing any charts upon which this chart depends.
  crds/               # Custom Resource Definitions
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

*Figure 2. Chart structure of a WordPress application (from Helm webpage[13])*

For the sake of clarity, up to this point, templates have been defined as manifests in which values are substituted by placeholders, which are populated at deployment time with default values specified in an external file, if not indicated otherwise. It should be mentioned that templating supports other objects, like the use of functions (i.e., operations over provided values, coming from Go and Sprig libraries), pipelines to chain functions, built-in objects (so placeholders can be substituted by the figures provided in values manifest, or by other pre-defined objects), flow control structures (i.e., if/else, with, range), variables, *named templates* (e.g., *define*, *template* and *include* objects, which can be defined in another part of the manifest or in an external one and re-used in other places), etc. Additional information and guidelines about the preparation of charts and their internal structuring for the ASSIST-IoT enablers is provided in Section 4.1.

### 3.1.3. Guidelines and structure for Helm packages

Once the components of an enabler are built, the associated k8s resources manifests have to be prepared (per component) and wrapped altogether in a compressed file, following a structure similar to the one presented in Figure 2. The generation of k8s templates can be done manually, using placeholders on them or specifying all key-value pairs in the templates (being plain k8s manifests in this latter case). There are different structures that the project enablers' charts could follow: (i) all the necessary templates of an application (in this case, of an enabler) within a single chart; (ii) a master or *umbrella* chart, with a separated chart per each component; or (iii) a hybrid approach, where the umbrella chart holds some of the components while the rest are maintained separately, as one can observe in Figure 3.

[13] https://helm.sh/docs/topics/charts/

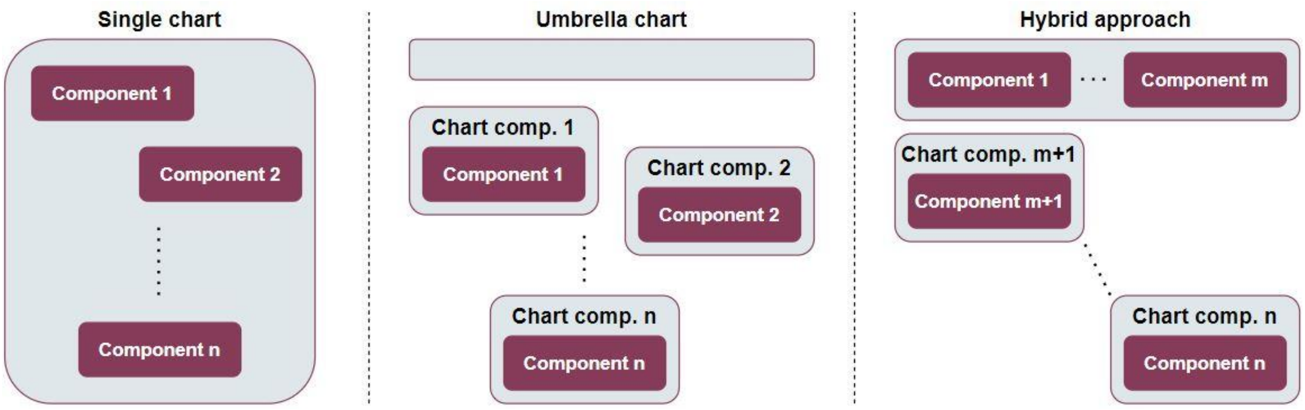
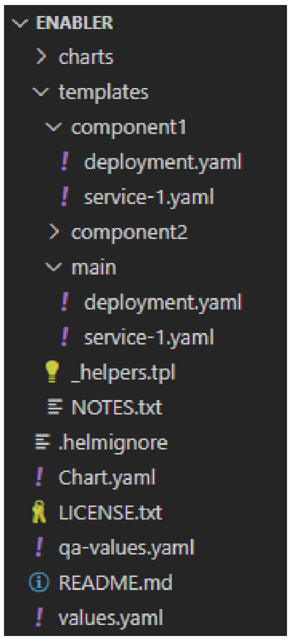


*Figure 3. Helm chart structuring strategies*

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. Still, for the sake of clarity, a **hybrid approach** (third option above) is encouraged for the enablers developed within the project: on the one hand, **custom project components** (including the main entrypoint, generally an API), or external yet adapted ones, will be placed **in the umbrella chart**. This means that their respective resource template manifests will be included in the template folder of this main chart; on the other hand, **third-party charts** used by the enabler will be included as **dependencies** (sub-charts), part of the enabler but not maintained by the project. An example of the latter could be an SQL database component (i.e., MariaDB chart) of an enabler. Hereinafter are provided a set of minimum guidelines aiming at easing the preparation, structuring and packaging of an enabler's chart. 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.

A **chart structure** adapted to the needs of ASSIST-IoT enablers has been developed to facilitate and speed up the development of charts. The idea of this structure is that developers/maintainers shall modify the default values of the *values.yaml* manifest, relaxing as much as possible the need of modifying the involved k8s templates. In any case, manual editing of the templates (e.g., adding more fields) or including additional resource descriptors (e.g., ConfigMap, PersistentVolumeClaim, etc.) is expected, as the necessities in terms of Kubernetes resources will be very varying for each enabler.

This structure consists mainly of the files and folders presented in Figure 4. It is essentially the structure presented in Section 3.1, but considering a **sub-folder per component** in the templates folder. A walkthrough over the folders, files and agreed conventions in the different manifests is provided:



```
ENABLER
  charts
  templates
    component1
      deployment.yaml
      service-1.yaml
    component2
    main
      deployment.yaml
      service-1.yaml
    _helpers.tpl
    NOTES.txt
  .helmignore
  Chart.yaml
  LICENSE.txt
  qa-values.yaml
  README.md
  values.yaml
```

*Figure 4. Proposed enablers' chart structure*

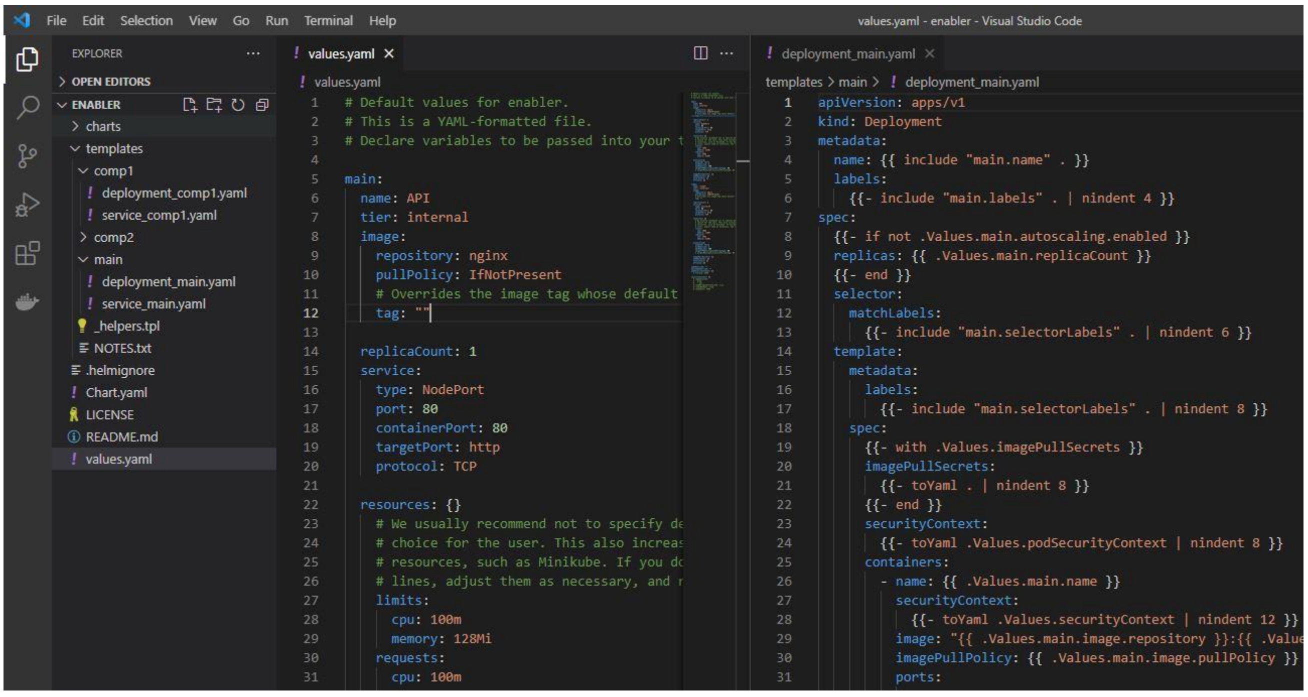
1. **Chart.yaml manifest**, which contains basic information about the chart. It can be composed of the fields presented in Figure 2, and among them, ASSIST-IoT enablers must include at least the following:
    - *apiVersion*: Helm API version (v2 for charts that require Helm 3).
    - *name*: name of the enabler.
    - *version*: version of the package. It must follow Semantic Versioning 2 (SemVer2[14]) conventions.
    - *description*: a few lines describing the enabler.
    - *dependencies*: list of external, non-maintained charts that are required by the enabler (must refer to its name, version and repository URL).
2. Generic **LICENSE** and **README** files will be included and have to be adapted accordingly to the respective rights (see Section 3.3) and functionalities of the enabler. The README file may include standalone instructions or point to a dedicated section in external wiki[15] (see also deliverable D6.5).
3. The **charts folder** will contain the templates related to each sub-chart/dependencies.
4. The **templates folder**, as aforementioned, will include a **sub-folder for each enabler component**. The following templates are required in each of them:
    - Workload template. It will be one of the following resources: Deployment, StatefulSet, DemonSet, CronJob or Job. Some fields must be present:
        - *apiVersion*: referred to k8s API version.
        - *kind*: type of k8s resource, in this case must be workload-related.
        - *metadata*: it can include multiple options; for ASSIST-IoT enablers, it requires the **name** of the **enabler component** and a set of **labels**, namely: *component* (label to group the replicas of this component), *enabler* (label to group all the k8s objects related to an enabler), *tier* (*internal* or *external*, in case it can be accessible from outside the ASSIST-IoT's clusters), and *isMainInterface* (yes or no).
        - *spec.replicas*: it specifies the minimum replicas required.
        - *spec.selector.matchLabels*: it specifies the label that will be used to manage the pods of the component. It should match to the label *component* defined in metadata.
        - *template*: it provides information related to the workload itself. It must include:
            - *metadata.label*, which must match with the aforementioned *components* label.
            - *spec.containers*, which must specify a *name*, point to the container *image*, specify its exposed *ports* and the required and limit *resources* (RAM, CPU) per pod. In case the containers have liveness and readiness probes, these should be specified as well.
        - *spec.nodeSelector* and *spec.affinity*. They provide a set of instructions to determine the node of the cluster in which the workload pods will be deployed.
    - Service template. This resource groups all the replicas of a workload into a common, exposed IP, providing also some load balancing features and facilitating discovery. Some fields must be present:
        - Analogously to the workload, it must include: *apiVersion*, *kind* (Service) and *metadata* (same labels as above).
        - *spec.type*: it should be one of the following: ClusterIP (if the workload is not or should not be accessible from outside the k8s cluster), NodePort or LoadBalancer (if the workload is deployed in the infrastructure of a Cloud provider).
        - *spec.selector*: it must point to the label *component* of the related workload.
        - spec.ports: it specifies the port of the external *service* (port), *protocol* and *name*.

[14] https://semver.org/
[15] https://assist-iot-enablers-documentation.readthedocs.io/

- Other resources might be included according to the nature of the component (e.g., ConfigMap, Secret, PersistentVolume, PersistentVolumeClaim, etc.).

5. The **values.yaml** manifest will specify the default values of the Kubernetes templates belonging to the enabler components. Most of the fields of the aforementioned templates will have placeholders to be substituted accordingly; still, in case that additional templates (for describing other resources) or fields in those descriptors are required, they should be prepared thereby.
6. The **qa-values.yaml** is a separated manifest, for testing in development/staging environments.

All the values of all the templates (and from the sub-charts) must be modified from the main values manifest. In the following example (Figure 5), an enabler is composed of three components, each of them with a Deployment and a Service manifest (there could be others). Most values of the objects belonging to these templates have placeholders (see the Deployment manifest below), which default values are indicated in the separated values.yaml manifest. These values can be modified at launch time, and the default ones can be changed by modifying the values manifest.



```yaml
# Default values for enabler.
# This is a YAML-formatted file.
# Declare variables to be passed into your t

main:
  name: API
  tier: internal
  image:
    repository: nginx
    pullPolicy: IfNotPresent
    # Overrides the image tag whose default
    tag: ""

  replicaCount: 1
  service:
    type: NodePort
    port: 80
    containerPort: 80
    targetPort: http
    protocol: TCP

  resources: {}
    # We usually recommend not to specify de
    # choice for the user. This also increas
    # resources, such as Minikube. If you do
    # lines, adjust them as necessary, and r
    limits:
      cpu: 100m
      memory: 128Mi
    requests:
      cpu: 100m
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "main.name" . }}
  labels:
    {{- include "main.labels" . | nindent 4 }}
spec:
  {{- if not .Values.main.autoscaling.enabled }}
  replicas: {{ .Values.main.replicaCount }}
  {{- end }}
  selector:
    matchLabels:
      {{- include "main.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
        {{- include "main.selectorLabels" . | nindent 8 }}
    spec:
      {{- with .Values.imagePullSecrets }}
      imagePullSecrets:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      securityContext:
        {{- toYaml .Values.podSecurityContext | nindent 8 }}
      containers:
        - name: {{ .Values.main.name }}
          securityContext:
            {{- toYaml .Values.securityContext | nindent 12 }}
          image: "{{ .Values.main.image.repository }}:{{ .Value
          imagePullPolicy: {{ .Values.main.image.pullPolicy }}
          ports:
```

*Figure 5. Example of template and values manifests*

A custom-made **chart generator** for the project was developed to create a baseline chart structuring for the enablers. Among other input, this software takes as input the number of components that an enabler has in order to create a folder with the set of predefined manifests (deployment and service for now), and specifies a separated object within the values manifest to declare its default values. Currently, the program only generates deployment as workload type, although it is expected to support additional ones in the future; in addition, the possibility of specifying other k8s resources via inputs of the script (PersistentVolumeClaims, Ingress, etc.) is also envisioned. In any case, it is very likely that developers will nonetheless perform manual modifications over the provisioned manifests, as they cannot foresee or be adapted to all the possibility (e.g., a component may have two ports exposed on its service-related manifest, so the second needs to be added manually).

## 3.2. Artifacts repositories

This section introduces the repositories that will be used for hosting the artifacts produced and/or adapted within the scope of the project. For the sake of completeness, not only the repositories related to packaged artifacts (i.e., enablers) will be listed, but also those related to code and containers.

As depicted in the DevSecOps methodology deliverable (see D6.1), **GitLab** will be 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. Apart from code repository, Gitlab will be also the DevSecOps open platform, enabling the execution of scripts for build automation and continuous integration, among others. Towards the end of the project, once the enablers are ready, the code repository will be migrated to **GitHub**, making them available for the rest of the community (hence, moving from a private to a public code repository).

Regarding containers, a similar strategy will be followed. **GitLab container registry** will be used during development phases 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**, under the umbrella of an ASSIST-IoT repository. This has been selected as it is the most used and extended place to share virtualized software artifacts with the rest of the community.

Finally, charts will also need a dedicated repository. Again, **GitLab package registry** will be used to host the charts developed in the project during its execution phase. 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**, under the umbrella of an ASSIST-IoT repository. Again, the selection responds to its common use in the community with respect to other alternatives. A summary of the repositories to be used is provided in Figure 6.

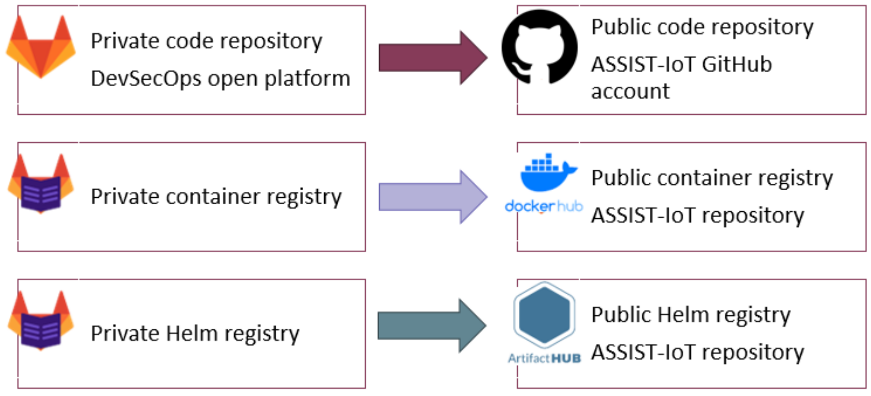


*Figure 6. Artifacts repositories*

## 3.3. Licensing

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.

Apache license 2.0 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.

Another open-source licenses are the MIT, GNU GPL and BSD, which are very similar to Apache 2.0. In the case of MIT license, a major difference with Apache 2.0 is that it mandates users to state any significant changes to the original code. It is not obligatory for users to reveal their source code, but it must include notifications about modifications. The BSD license is almost identical to MIT, being the main difference that the use of the name of the original creator requires of authorisation, permitting creators to protect themselves from associations with a defective or poorly-written software. Regarding GNU GPL, software that uses any GPL-licensed component has to release its full source code and all rights to modify and distribute the entire code.

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. Additional information will be provided in the final report associated to Task 9.4 (D9.7).

# 4. Release Cycle

The release cycle comes to play once the functions of the components are ready, validated and built, i.e., once all their associated requirements have been fulfilled and the containers are ready. It has been divided in 2 parts: the packaging of the solution (in this case, of an enabler with its components), and its configuration and deployment, considering Helm as the main technology involved (alongside Docker and Kubernetes).

It is important to remind that the release part itself in a usual DevOps cycle (as well as in the project’s DevSecOps approach) just entails the instantiation of the packaged solution from an artifact repository in a replica of a development environment for its acceptance. As this deliverable encompasses the packaging-related activities, the cycle is expanded to cover aspects from the packaging of enablers to their configuration and deployment, so they can be validated in a staging environment (the building process is not included).

## 4.1. Packaging

In the previous section, the main technology to be leveraged for the packaging process were introduced, alongside the rationale behind its need and selection. Also, a methodology is provided to construct these packages, known as charts. In turn, this section introduces the main commands involved in the packaging process, aiming at **preparing dedicated scripts** to be later on introduced within the whole DevSecOps pipeline (hence, not prepared so far). In this way, the packaging process can be automated to some extent. The key commands that must be part of the packaging process (either manual or automated), once a chart structure is ready, are the following:

- helm **dependency update**. This command verifies that the charts present in the namesake folder are those required in the Chart.yaml manifest. If these are not present or the version is wrong, it will download the required charts from the repositories indicated in such manifest.
- helm **package**. This command is in charge of packaging a chart into a versioned, compressed chart file.
- helm **push**. Used for uploading a packaged chart to a registry, requiring an URL. Depending on the registry, login or token credentials should be specified. This command should point to an ASSIST-IoT Helm repository, concept introduced in Section 3.2.

Other commands may be needed when preparing an enabler chart, more information about other available commands can be found in Helm documentation[16]. Still, this document does not act as a full Helm manual and knowledge of use is required (for instance, helm **repo add/update**, helm **search repo**, helm **show chart** are commands needed to manage repositories, search charts on them, get information from their charts, etc.). The packaging itself is a simple process, however, a good internal structure should be followed for later understanding and sharing of the enabler. Thus, following the guidelines specified in Section 3.1.3 for preparing the charts is encouraged. The flow is summarised in Figure 7.

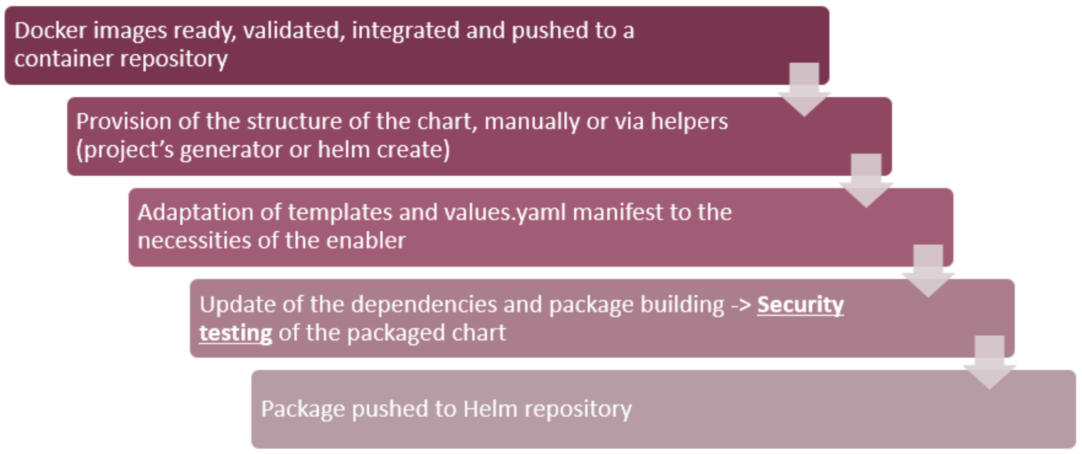

Docker images ready, validated, integrated and pushed to a container repository
Provision of the structure of the chart, manually or via helpers (project’s generator or helm create)
Adaptation of templates and values.yaml manifest to the necessities of the enabler
Update of the dependencies and package building -> Security testing of the packaged chart
Package pushed to Helm repository

*Figure 7. Packaging flow*

[16] https://helm.sh/docs/helm/helm/

It is important to remember that security has to be integrated in the DevOps process, to effectively achieve a suitable DevSecOps methodology. In this project, different open-source security tools have been assessed to be included within the DevSecOps pipeline. The release cycle does not focus on code or integration aspects (neither security nor non-related), but rather on packaging ones. For this reason, security tools focused on evaluating integrity, formatting and security aspects of the developed Helm charts are presented:

- **KubeLinter**[17]: This tool evaluates Kubernetes manifests and charts and checks them against a variety of best practices, focusing on production readiness and security. The provided reports can help teams to check regularly security misconfigurations and DevOps-related best practices. Some functionalities that it checks are ensuring that containers will run as a non-root user, enforcing least privilege, and that sensitive information is only stored in secrets.
- **Checkcov**[18]: It is a static code analysis tool for infrastructure-as-code. It scans cloud infrastructure provisioned using Terraform, AWS SAM, Kubernetes, Helm, Kustomize, Dockerfile, or ARM Templates (among others) and detects security and compliance misconfigurations using graph-based scanning. In the particular case of Helm (hence, need of specific use within the ASSIST-IoT pipeline), it has more than 150 out-of-the-box security and misconfiguration checks for manifests.

In the scope of the project, **KubeLinter** will be the security testing tool implemented by default within the DevSecOps workflow, as it is a tool dedicated and maintained specifically for Helm charts and has dedicated guidelines to be used with integration tools.

## 4.2. Configuration and deployment

The configuration is a critical part of any software artifact, especially when some or many of them must communicate among themselves to provide a more complex service or have to be instantiated in environments with specific constraints (e.g., the port of a specific service is already in use). A Helm chart holds its default configuration in the *values.yaml* manifest, as all the environment variables of the underlying containers, that should be modifiable by a user, are (or should be) specified in there. It should be mentioned that this manifest can include many other aspects of the chart, not just environment variables of containers: labels, type of exposition (to outside the cluster or just within), update strategy, number of replicas, assigned hardware resources, volume locations, container image or source repositories, etc.

The default configuration specified in the chart can be modified in different ways:

a) When instantiating a chart, 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.

b) When instantiating a chart, 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.

c) Modifying manually the *values.yaml* manifest, once downloaded and edited (then, wrapped again), prior to its installation, not being a common approach.

As expected, helm **install** is the command used for deploying a chart. Still, in the scope of the project, although it can be considered and **is important** to know **for testing/staging/integration** environments, it will **not** be the main mechanism **for user operations** (although it will be present underneath). Users will interact either graphically via the manageability enablers, or via the API of the smart orchestrator (see D4.2 and D5.3).

To **configure and deploy an enabler via the smart orchestrator API**, the user must use the endpoint /api/enablers (POST command). In case of using it, custom values of the *values.yaml* manifest can be passed using the body of the message, particularly via the *additionalParams* object. Prior to that, the user should have made use of additional endpoints, to set up the clusters managed by the system and to add Helm repositories (among others). More information can be consulted in the project wiki[19].

[17] https://docs.kubelinter.io/#/
[18] https://www.checkov.io/
[19] https://assist-iot-enablers-documentation.readthedocs.io/

In contrast, **manageability enablers** act as intermediaries between users and the smart orchestrator API, facilitating the overall flow thanks to their graphical interfaces. To avoid duplicating data along documents, information about its usage can be found both in D6.5 and in the project wiki[12]. As a pending action under this task, the project aims at providing some templates to specify the values that could be modified by a user, so that she/he can set them via form with options. The configuration and deployment flow is summarised in Figure 8.

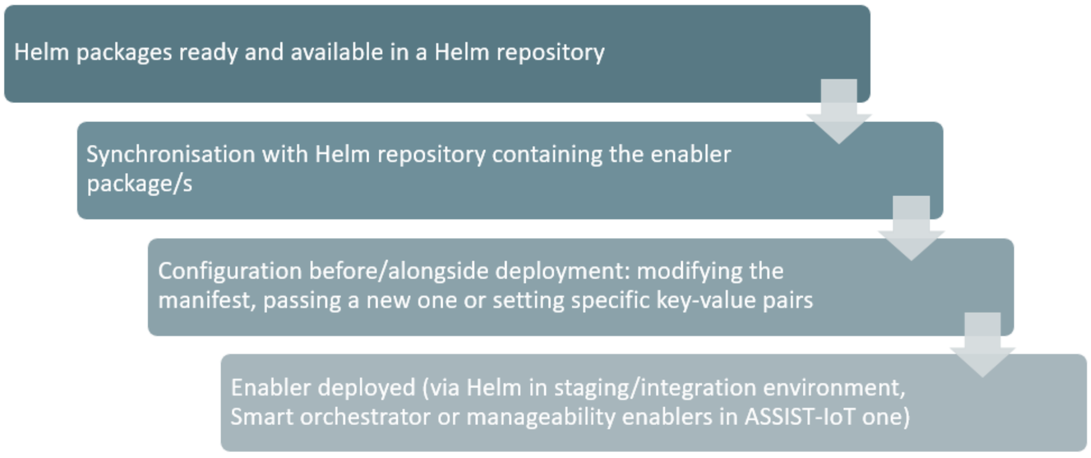

Helm packages ready and available in a Helm repository
Synchronisation with Helm repository containing the enabler package/s
Configuration before/alongside deployment: modifying the manifest, passing a new one or setting specific key-value pairs
Enabler deployed (via Helm in staging/integration environment, Smart orchestrator or manageability enablers in ASSIST-IoT one)

*Figure 8. Configuration and deployment flow*

# 5. Release Plan

The ASSIST-IoT project will release a large number of enablers (~40), providing functionalities related to horizontal planes or verticals of the reference architecture. Each enabler follows the DevSecOps methodology, as specified in Section 2. 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 (leveraging Docker compose or generating k8s manifests) and *package* them (via preparing their Helm charts). Enablers widely vary in complexity and criticality. Hence having a unified release plan for all the enablers of the project is not suitable.

The final action that has to be performed under the scope of the release plan is the execution of **acceptance tests** in a **staging environment**. A staging environment is a replica of a production environment for software testing, being the last step before assuring that enablers are ready (in this case, for being used in the pilots – WP7). This environment is common for the action, and it is the place where most of the tests will be developed. Tests are executed under the umbrella of T6.2, which planning and outcomes are depicted in D6.2 and D6.3, respectively. In the particular case of acceptance testing, they should be performed for the enablers as a whole, once packaged, to validate them before production. This acceptance will be performed, considering the tests specified in D6.2. Any bug found on this stage may require an update on the code, building or packaging of an enabler.

ASSIST-IoT has defined two major platform-level release dates. The **first release** of the project is envisioned for M18 (April 2022). This (platform-level) release will contain a functional version of the **essential enablers**, namely: smart orchestrator, long-term storage enabler, edge data broker, VPN enabler, tactile dashboard, Open API, identity manager, authorization enabler, DLT logging enabler, and manageability enablers. Some of them will be already packaged, with their respective charts tagged following Semantic Versioning 2 (>=v1.0.0) as aforementioned. As working with Helm requires specific knowledge, some of their charts might be slightly delayed as some partners are not that familiar with this technology. Some additional considerations:

- This release will be accompanied by a script to facilitate the deployment of a set of these enablers in a top-tier node (see D6.5 – installation script).
- A set of guidelines and recommendations for setting-up the infrastructure topology and some software pre-requirements.
- Other non-essential enablers may have already a functional first release for this first release.
- All the released enablers will have configuration, deployment and usage guidelines in their respective wikis, among other information.

The **second release** is expected for M27 (January 2023). This date matches with the beginning of the second Open Call of the project. This is important because, while for the first round of external third-parties to join the project (Open Call #1 winners) will validate enabler under mostly essential enablers in testing/isolated/hybrid conditions, the second round of parties will actually interact with the complete platform and the whole list of enablers in a "close-to-production-ready" fashion. Therefore, all the enablers of the project must have a functional, packaged version ready. It should be highlighted that:

- Non-essential enablers may likely have their first ASSIST-IoT-ready release (i.e., Helm chart ready) between both release phases.
- Some essential enablers might not have a second major release, just minor or patch versions. It depends on the degree of functionalities provided in the first one (usually related to incompatible API changes). On the contrary, enablers may suffer several major versions between both project-level release dates.

Developments are expected until M30 (April 2023). The period from the second release will serve different purposes: (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/or (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. A summary is depicted in Figure 9.

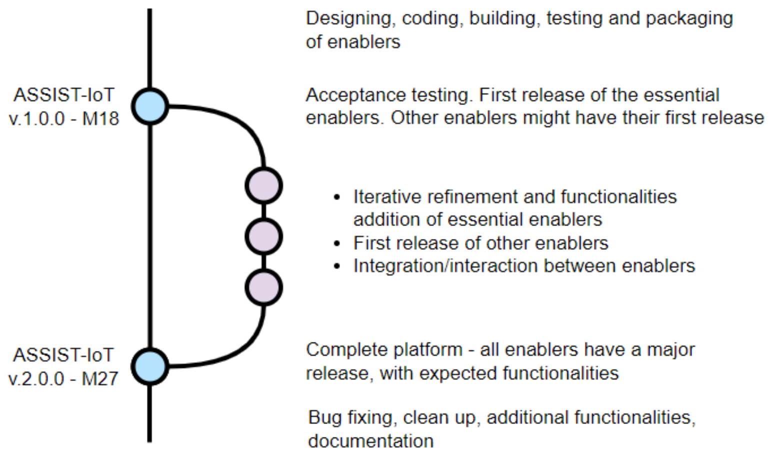

Designing, coding, building, testing and packaging of enablers
ASSIST-IoT v.1.0.0 - M18
Acceptance testing. First release of the essential enablers. Other enablers might have their first release
• Iterative refinement and functionalities addition of essential enablers
• First release of other enablers
• Integration/interaction between enablers
ASSIST-IoT v.2.0.0 - M27
Complete platform - all enablers have a major release, with expected functionalities
Bug fixing, clean up, additional functionalities, documentation

*Figure 9. Release plan of ASSIST-IoT platform*

# 6. Future work

This deliverable has presented the packaging and releasing methodology, cycle and plan to be followed by the enablers developed within the ASSIST-IoT project. More specifically, the following content is provided within this document:

- Positioning of packaging and releasing within the DevSecOps cycle,
- Introduction of packaging tools and selection of Helm,
- Artifact repositories to be considered for ASSIST-IoT development and sharing,
- Licensing strategy,
- Packaging, configuration and deployment (Helm) commands,
- Release planning for enablers (mostly related to dates and versioning).

This deliverable corresponds to the first document of a series of two iterations. In the following one, additional data related to the outcomes will be included, including:

- Information about the final project's Helm chart structure generator,
- Implementation of DevSecOps pipeline, focusing on the automated scripts for:
  - Packaging enablers,
  - Uploading software artifacts (code, containers and charts) into public/private repositories,
  - Security checking.
- Information about the final licenses applied to enablers and their components,
- Lessons learned around the release and distribution strategy,
- And report over the stickiness to the plan.