



Deliverable D1.6

DevOps methodology and CI/CD strategy for EMERALD-v2

Editor(s):	Gorka Benguria, Iñaki Etxaniz (TECNALIA)
Responsible Partner:	TECNALIA Research and Innovation
Status-Version:	Final - v1.0
Date:	30.04.2025
Type:	R
Distribution level (SEN, PU):	PU

Project Number:	101120688
Project Title:	EMERALD

Title of Deliverable:	D1.6 DevOps methodology and CI/CD strategy for EMERALD-v2
Due Date of Delivery to the EC	30.04.2025

Work package responsible for the Deliverable:	WP1 - Concept and methodology of EMERALD
Editor(s):	Gorka Benguria Elguezabal, Iñaki Etxaniz (TECNALIA)
Contributor(s):	Gorka Benguria Elguezabal, Iñaki Etxaniz (TECNALIA)
Reviewer(s):	Franz Deimling (FABA) Cristina Martínez, Juncal Alonso (TECNALIA)
Approved by:	All Partners
Recommended/mandatory readers:	WP1, WP2, WP3, WP4, WP5

Abstract:	Final version of the description of the DevOps methodology and CI/CD strategy that provides details on the integration process followed to create and deploy the integrated EMERALD CaaS (Compliance as a Service) Framework. It also provides details on the strategies applied at integration and deployment level to help on the achievement of the EMERALD goal.
Keyword List:	DevOps, CI/CD, Integration, Container, Environment, Releases
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0 DEED https://creativecommons.org/licenses/by-sa/4.0/)
Disclaimer	Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. The European Union cannot be held responsible for them.

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	15.04.2025	First draft version	TECNALIA
v0.2	17.04.2025	Typos corrected and style polished. Sent for internal QA review	TECNALIA
v0.3	17.04.2025	QA Review	Franz Deimling (FABA)
v0.4	26.04.2025	Addressed all comments received in the Internal QA review	TECNALIA
v0.5	28.04.2025	Final review	TECNALIA
v1.0	30.04.2025	Submitted to the European Commission	TECNALIA

Table of contents

Terms and abbreviations.....	7
Executive Summary.....	8
1 Introduction	10
1.1 About this deliverable.....	10
1.2 Document Structure	11
1.3 Updates from D1.5.....	11
2 DevOps Methodology	14
2.1 Context.....	14
2.2 Goals	14
2.3 Processes	15
2.3.1 Plan	16
2.3.2 Code.....	17
2.3.3 Build.....	18
2.3.4 Test	18
2.3.5 Release.....	19
2.3.6 Deploy.....	20
2.3.7 Operate.....	20
2.3.8 Monitor.....	21
2.4 Lifecycle	21
3 CI/CD Strategy	24
3.1 CI Strategy.....	24
3.1.1 Container-based	24
3.1.2 Environments with IaC	25
3.1.3 Integration guidelines.....	26
3.1.4 CI/CD Components	26
3.1.5 Component-based Kustomize	29
3.1.6 Manual deployment support.....	30
3.1.7 Rancher for debugging support.....	32
3.1.8 Local environment for testing	32
3.1.9 Progressive Verification.....	32
3.1.10Automation.....	33
3.2 CD Strategy	33
3.2.1 Releases	34
3.2.2 Public Assets Release.....	34
3.2.3 Keycloak configuration	35
3.2.4 Demo pilot	35

3.2.5 Documentation.....	35
3.2.6 Environments with IaC	36
3.2.7 Automation.....	36
4 Conclusions	37
5 References.....	38
APPENDIX A: Project Risks and impact in the DevOps Methodology	40
APPENDIX B: Project Milestones from the DoA	42
APPENDIX C: Integration files.....	43
C.1 – Renovate mechanism	43
C.2 – Semantic Versioning Configuration	44
C.3 – Kustomize approach	45
C.4 – Docker compose approach.....	48
C.5 – CI/CD Examples.....	50

List of figures

FIGURE 1. DEVOPS CYCLE	16
FIGURE 2. LIST OF ISSUES RELATED TO CONCEPT & METHODOLOGY IN EMERALD.....	22
FIGURE 3. A MERGE REQUEST RELATED TO CONCEPT & METHODOLOGY IN EMERALD.....	22
FIGURE 4. A MERGE REQUEST MECHANISM TO PRODUCE A NEW RELEASE IN EMERALD.....	34
FIGURE 5. RENOVATE SCHEDULE	44
FIGURE 6. KUSTOMIZE MAIN STRUCTURE OF THE INTEGRATED CAAS FRAMEWORK.....	45
FIGURE 7. KUSTOMIZE COMPONENT	47
FIGURE 8. DOCKER COMPOSE FRAMEWORK	49
FIGURE 9. LOCAL ENVIRONMENT SERVICES	50
FIGURE 10. RCM CHANGE	50
FIGURE 11. DOCKER CI/CD EXAMPLE	51
FIGURE 12. DOCKER CI/CD EXAMPLE PIPELINES.....	52
FIGURE 13. DOCKER CI/CD STAGES DETAIL.....	52
FIGURE 14. SEMANTIC RELEASE CI/CD EXAMPLE	53
FIGURE 15. SEMANTIC RELEASE CI/CD STAGES DETAIL.....	54

List of tables

TABLE 1. OVERVIEW OF DELIVERABLE UPDATES WITH RESPECT TO D1.5.....	12
TABLE 2. RISK AND MITIGATION LIST	40

List of listings

LISTING 1. CI/CD FOR THE RCM COMPONENT	29
LISTING 2. CI/CD FOR SIDE-SERVICE	29
LISTING 3. EXAMPLE COMMANDS TO MANUALLY REDEPLOY A COMPONENT	31
LISTING 4. RENOVATE PIPELINE (.GITLAB-CI.YML)	43
LISTING 5. CONTENT OF THE FILE RENOVATE.JSON.....	44

LISTING 6. CONTENT OF THE FILE .RELEASERC.YAML.....	45
LISTING 7. KUSTOMIZE INTEGRATE OVERLAY	46
LISTING 8. KUSTOMIZE BASE	46
LISTING 9. KUSTOMIZE RCM COMPONENT	48
LISTING 10. CI/CD FOR DOCKER GENERATION	51
LISTING 11. CI/CD FOR SEMANTIC RELEASE GENERATION	54

Terms and abbreviations

AI	Artificial Intelligence
AI-SEC	AI Security Evidence Collector
AMOE	Assessment and Management of Organizational Evidence
API	Application Programming Interface
CaaS	Compliance-as-a-Service ¹
CI/CD	Continuous Integration / Continuous Deployment
CLI	Command Line Interface
CMMI	Capability Maturity Model Integration
DevOps	Development and Operation
DIND	Docker in Docker
DoA	Description of the Action
EC	European Commission
EUCS	European Cybersecurity Certification Scheme for Cloud Services
GA	Grant Agreement to the project
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ITIL	Information Technology Infrastructure Library
K8s	Kubernetes
K8so	Kubernetes on Openstack
K8sv	Kubernetes on Vsphere
MARI	Mapping Assistant for Regulations with Intelligence
OSCAL	Open Security Controls Assessment Language
RCM	Repository of Controls and Metrics
TWS	Trustworthiness System
UI/UX	User Interface / User Experience
WP	Work Package

¹ Please note that in previous deliverables and in the DoA, the term Certification-as-a-Service was used to stand for CaaS. Compliance has now been introduced to clarify that EMERALD can be used to assess both normative models and internal organizational models.

Executive Summary

The main objective of EMERALD project is to provide a framework that enables continuous compliance and certification, and agile lean re-certification to consume services that adhere to a defined level of security and trust in a uniform way across heterogeneous environments made of combinations of various resources.

In order to achieve the development of the CaaS (Compliance-as-a-Service)² Framework, the EMERALD project aggregates specialized components from different teams and integrates them into a single framework that is validated in different pilots. This brings some challenges that should be managed: on the provider side, the components are developed by different teams with different schedules and different development practices; on the consumer side, the CaaS Framework will be consumed as a service or as an on-premises configuration.

This document is the final version of the DevOps (Development and Operations) Methodology and the CI/CD (Continuous Integration/Continuous Deployment) strategies. The previous version, presented at M6, focused mainly on establishing the corresponding processes and tools to achieve the first integrated version of the CaaS Framework (M18). This final version is the result of the experience and lessons learnt during the first half of the project and focuses on the upcoming releases of the CaaS Framework (M30 and M34) and the deployment of the CaaS Framework in the pilots. The main changes that have been introduced relate to the following aspects:

- **Integration of new versions of the CaaS Framework:** Supporting, on the one hand, the agile integration of components developed by the different teams and, on the other hand, the validation of the upcoming versions of the CaaS Framework before being deployed in the pilots with a certain degree of confidence.
- **Delivery of new versions to the Pilots:** Supporting the deployment of the validated versions of the CaaS Framework in the pilots and introducing mechanisms to reduce the undesirable effects of deployment in their environments (such as data loss, the need to reconfigure the environment or service downtime, to name a few).
- **Getting and processing feedback:** Supporting the collection of feedback from the pilots and the development teams, and the processing of that feedback to improve the CaaS Framework.

The target audience of the document are the EMERALD participants in charge of coordinating the development and operation activities. In addition, the document also aims to provide information to other partners and stakeholders in understanding how the EMERALD CaaS Framework is managed from a DevOps perspective.

The document also includes annexes that provide additional information about the project, which can help the reader grasp the project context. In particular, they include the risks and milestones defined in the Description of Action (DoA) and details on the integration files that are not accessible in the public area of the GitLab repository of the EMERALD project.

This document is the second and final version of the development and operation coordination approach that has been applied during the first half of the project. Future related work will involve applying the techniques defined here to develop the v2 (M30) and v3 (M34) releases of the CaaS Framework, as well as to the deployments in the pilots. Some changes to the

² Please note that in previous deliverables and in the DoA, the term Certification-as-a-Service was used to stand for CaaS. Compliance has now been introduced to clarify that EMERALD can be used to assess both normative models and internal organizational models.

methodology are likely to be introduced, but the fundamental processes are already defined and in operation.

1 Introduction

This section introduces the context of the project, the aim and audience of the content and the document structure.

This deliverable is the result of task T1.3 *Continuous integration and optimization* and details the line of actions to be followed in the project to achieve a smooth and quick transition among the developers' work and the final result, i.e. the EMERALD CaaS Framework deployed in the pilots.

This document, D1.6, is the successor of D1.5 [1], which presented the first version of the development and operation coordination approach and was applied as a baseline during the first stage of the project. D1.6 follows the same structure and preserves part of content from the previous version of the deliverable, in order to keep the document self-contained and easier to follow. In this sense, section 1.3 presents the main modifications of this document compared to its first version.

1.1 About this deliverable

From the project mission:

“EMERALD’s mission is to provide a user-friendly framework to help stakeholders in the cybersecurity field efficiently manage certifications, enhancing the security and effectiveness of cloud service usage. The proposed EMERALD environment will be the foundation for defining a new service for assisting the certification process that we named Certification-as-a-Service (CaaS).” [2]

To contribute to that mission, this deliverable describes the technical coordination approach that is followed in the EMERALD project to continuously integrate, update, and validate that framework, here in after referred as Compliance-as-a-Service² (CaaS) Framework. The elements of the approach have been defined considering the context of the EMERALD project and the envisaged CaaS Framework, which are detailed in following sections.

In this document we describe the two main elements of the technical coordination approach: the DevOps Methodology, and the CI/CD Strategy:

- The DevOps Methodology in EMERALD focuses on how the development, integration and validation teams collaborate to build and evolve the CaaS Framework through its planned releases to achieve the evolving requirements considering the resources and constrains of the EMERALD project.
- The CI/CD Strategy describes the technical approaches to be applied to continuously integrate and deploy the evolving components and how these integrated versions are deployed and validated in the target consumers. The continuous integration (CI) deals with the integration of the different components that build up the CaaS Framework and the verification of the integrated version, so that it is ready for its deployment. The continuous deployment (CD) deals with the deployment of the integrated version in the production environment and the required pilot environments for their validation. Within the continuous deployment we also establish the mechanisms to manage the feedback into new or extended requirements.

The target audience of this document is:

- DevOps engineers of the EMERALD project. For them, this document provides a guide for managing the integration of the technical outcomes of the development teams, as well as for the deployment of the integrated versions for their validation.

- Developers aiming to integrate or update the components of the CaaS Framework. For them, this document provides a guide to understand how to integrate their components in the CaaS Framework, and how to send their new versions for validation in order to generate new integrated versions of the CaaS Framework.
- CaaS Framework users. For them, this document provides a guide to understand how to manage the feedback from their usage so that it can be used to adjust the CaaS Framework to fulfil their needs. And, in case they want to deploy the CaaS Framework in their own environments, it provides them with the necessary information on how to deploy the CaaS Framework and how to manage the updates.
- Finally, this document is also targeted to people that want to understand how the CaaS Framework was developed. For them, it also provides some resources that are useful to extend or customize the CaaS to their needs.

This document is the final version of the DevOps methodology and CD/CI strategy for EMERALD. It is likely that some changes will be made to the methodology and the CI/CD Strategy, but the fundamental processes are already defined and in operation.

1.2 Document Structure

The document is organized in two main sections:

- DevOps Methodology
- CI/CD Strategy

The DevOps Methodology section, Section 2, explains the foundations for the approach that are used to coordinate the development and operation in order to continuously integrate, validate and manage feedback to improve the CaaS Framework, considering the needs of the EMERALD pilots and the contributions of the development teams.

The CI/CD Strategy section, Section 3, is split in two sections: CI Strategy and CD Strategy. The CI Strategy section describes the principles and practices used for the integration of the different components of the CaaS Framework prior to its disposal to the pilots. The CD Strategy section describes the approach and tools used to provide the integrated versions of the CaaS Framework to the pilots for their validation, as well as the mechanisms to collect that validation feedback for the continuous improvement of the upcoming CaaS Framework versions.

Finally, Section 4 presents the main conclusions of the document.

In addition, the document includes three annexes (*APPENDIX A: Project Risks and impact in the DevOps Methodology*, *APPENDIX B: Project Milestones from the DoA* and *APPENDIX C: Integration files*), which support the understanding of the document, as well as the technical implementation details of some specific assets of the DevOps infrastructure used.

1.3 Updates from D1.5

This deliverable evolves from D1.5 [1], and with the ultimate goal of making the document self-contained and easier to follow, some of the content comes from D1.5, as it is unchanged, and some is new. To simplify tracking progress and updates from the previous version (D1.5), Table 1 shows a brief summary of changes and additions to each section of the document.

The first version of the deliverable presented in M6 was mainly focused on the generation of the first integrated version of the CaaS Framework (M18). Since M6 many activities have been performed in the project:

- 33 development cycles have been carried out (one every two weeks, excluding holidays)

- 17 repositories have been created under the DevOps group in the EMERALD GitLab repository.
- 34 issues have been created in the DevOps group in the EMERALD GitLab repository.
- 66 merge requests have been created in the DevOps group in the EMERALD GitLab repository.
- Over 400 commits have been made in the DevOps group in the EMERALD GitLab repository.
- 7 patch releases of the CaaS Framework have been created.
- 24 patch releases of the Side Services have been created.

This final version is focused on the deployment of the CaaS Framework in the pilots, and its controlled evolution. In that sense, this document integrates the lessons learnt during the first half of the project and includes strategies to effectively support the pilots in the usage of the CaaS Framework while gathering feedback from them.

Table 1. Overview of deliverable updates with respect to D1.5

Section	Changes
2. DevOps Methodology	<p>The processes of the Methodology are described in more detail with the lessons learn.</p> <ul style="list-style-type: none">• The plan process introduces the bi-weekly periodic meetings that have been carried out since the beginning of the project.• The code process describes the support areas that have been provided to the development teams.• The test process describes the integration tests and the priorities that have been set for their implementation.• The release process describes the major and the patch releases that have been applied.• The deploy process describes the two types of deployment that are supported: SaaS and on-premises.• The operate process describes the feedback gathering and processing that is performed in the project.
3. CI/CD Strategy	<p>Minor updates on the previous strategies have been made and new strategies have been added.</p> <ul style="list-style-type: none">• Integration guidelines: Guidelines that have been provided to the development teams to help them in the integration of their components.• CI/CD Components: CI/CD components used in the project.• Manual deployment support: Manual deployment support provided to the development teams.• Rancher for debugging support: Rancher tool used to provide fast debugging support to the development teams.• Local environment for testing: Local environment used to provide local testing support to the development and DevOps teams.• Public Assets Release: Strategies used to release the public assets of the project.• Keycloak configuration: Identity and access management approach used in the project.• Demo Pilot: Approach to support pilots to be applied in the project.

Section	Changes
APPENDIX C: Integration files	New annex that provides low level details on some key processes applied during the integration activities: <i>Renovate</i> mechanisms, <i>Semantic Versioning</i> configuration, <i>Kustomize</i> approach, and <i>Docker Compose</i> approach. We also include some CI/CD samples as reference as they are placed in the internal GitLab area.

2 DevOps Methodology

This section describes the process, and the lifecycle applied in EMERALD to coordinate the development and operation teams during the project. The section also presents the challenges and risks faced by the EMERALD DevOps Methodology. From these challenges, we define some top-level goals for the DevOps Methodology to be applied. Then, we present the process to be applied, whose main tasks are specified. Finally, the software lifecycle is presented, that describes how the process is applied over time.

The description contains much information in common with D1.5 [1] with the final aim of providing a self-contained section that facilitates the reader's understanding.

2.1 Context

EMERALD presents some challenges and risks that should be managed during the CaaS Framework development. The Description of Action (DoA) of EMERALD [3] includes some risks that are relevant for the definition of the DevOps Methodology (see *APPENDIX A: Project Risks and impact in the DevOps Methodology* for more details and an extended list):

- Users experience low usability.
- EMERALD components are not able to be fully integrated.
- The implementation does not cover all use cases.
- Underestimation of effort needed to complete activities.
- Technology changes require significant redesign of the EMERALD architecture.
- A partner fails to meet the obligations and becomes non-performing or even defaulting.

Apart from these risks enumerated in the DoA, there are several challenges to be considered during the elaboration of the DevOps Methodology:

- We are aiming a TRL7 [4] “System/process prototype demonstration in an operational environment” (integrated pilot system level).
- We have different components with different requirement sets, different teams, and different agendas.
- We have fixed milestones (see *APPENDIX B: Project Milestones from the DoA*) at project level that should be achieved.
- The CaaS Framework must be deployed as a Service. That implies to integrate and test all the components in a production grade service environment.
- Some pilots, due to internal policies, may require deploying the framework for internal validation.

2.2 Goals

Based on the risks and challenges established, the DevOps Methodology we are aiming for should have the following characteristics:

- **Release-based:** We need a release-based methodology because the DoA states an iterative approach for the CaaS Framework, where at least three releases will be provided. Therefore, a minimum of three versions are expected, with some intermediate versions that can be motivated by other project milestones.
- **Manage the feedback:** The project aims a TRL7 outcome. That implies that at the end of the project, apart for being finished, the system must be validated in real life environments. The project has several validation cycles planned, and the methodology should keep track of the issues raised during these activities to make sure they are managed.

- **Manage the aimed component set.** To keep track of the integration of all the components, it is necessary to have a clear idea on how the components are integrated. Most of the components will be integrated in Kubernetes, but some of them (e.g., the evidence collectors) will run individually sending the information to the Framework.
- **Keep requirement traceability.** We use the requirements as the basis for the validation of the different components of the CaaS Framework, as well as the framework as a whole. Therefore, there should be a traceability of the DevOps activities with the set of requirements. The requirements are managed in GitLab and are reported in D1.3 [5] (M12), and D1.4 [6] that will be submitted in M24.
- **Manage the environments.** The EMERALD project is required to manage several environments during the DevOps activities. The project envisions at least two environments: integration and production, but additional ones could also be managed on demand. The Integration environment is focused on providing debugging support for the developers and verification means for the project, whereas the production environment is focused in providing a validation platform for the project. Additional environments can be created at any point by the pilots themselves or by the DevOps team, based on specific needs, and lasting for a variable timeframe.
- **Integrate as soon as possible.** The update requests from the diverse development teams should be promoted into the integration environment as soon as possible. This helps mitigate some of the risks identified, such as effort underestimation and partner withdrawal.

2.3 Processes

In order to select the set of processes to be used to support coordination between development and pilots, we have a wide range of standards and other kinds of references to choose from. From the standards side, there is no normative DevOps standard that can be used as a basis. The most approximate elements that can be found are standards for software development and for service operation maturity evaluation, such as:

- CMMI v1.3 [7], where we can find 22 process areas.
- ISO 15504 [8], where we can find 48 processes.
- ITIL [9], where we have 34 management practices.

From these standard references, interesting process categories can be extracted, such as architecture definition, monitoring, release, issue creation, validation, deployment, infrastructure, integration, requirements elicitation, training, management, improvement, configuration, etc.

Focusing on the DevOps process – as stated above – a review of the literature shows that, even if there are some propositions about a DevOps process model in [10], [11], there is no sound common standard [12], [13], [14], [15].

For the definition of the processes to be executed in the EMERALD DevOps Methodology, we have taken as a starting point the DevOps cycle described in many publications [16], [17], [18], [19], [20]. This cycle is also present throughout grey literature [21]. Figure 1 shows the basic structure of the DevOps cycle shared along those publications. The Cycle is composed of the following processes:

- **Plan:** During this process, the requirements are collected and prioritised. The requirements are then used to create issues that are planned for next iterations.

- **Code:** During this process, the integration of the different components is encoded as a container orchestration code (aka choreography). In addition, code for the environment and side-services can be developed, when needed.
- **Build:** The building process is automated. It takes the choreography and deploys the CaaS Framework in the development environment.
- **Test:** The test process focuses on the integration testing. It covers the development of the necessary tests to verify the behaviour of the framework in the long term.
- **Release:** We create a formal release of the CaaS Framework for each release milestone planned, as well as when updates in components require it.
- **Deploy:** The deploy process is automated. It takes the choreography and deploys the tagged CaaS Framework in the production environment, and the pilot environments are informed about the new version.
- **Operate:** Apart from the regular operation, this process is focussed on the validation management.
- **Monitor:** The monitoring of the production environment is adapted in this stage to cover the production and pilots' environments.

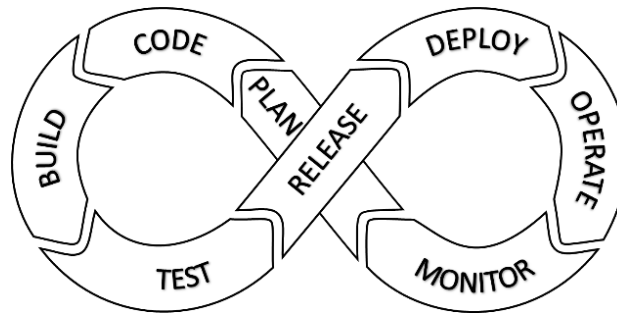


Figure 1. DevOps Cycle

We take these processes as a basis for our DevOps Methodology, adapting them to the specificities of the EMERALD project. For each process, we provide below a brief description and the expected inputs and outputs.

2.3.1 Plan

During the planning process, we collect the integration and deployment needs of the project, prioritize the activities to be performed, and track their completion. This activity is performed on demand and periodically.

Whenever a need over the integration is identified or received, it is collected, and a quick prioritisation is performed. In case the need is urgent, it is put into implementation. Otherwise, it is put into the backlog of needs to be performed in the next iterations.

Besides, every two weeks, in a meeting with the development teams and the pilots, we collect their needs and prioritise them. That meeting is also used to review the status of the tasks that are being worked out in the DevOps team.

The starting point for the needs of the project was the project DoA [3], where the EMERALD overall needs were described, and then the requirements gathered in the project architecture deliverables, D1.3 [5] and D1.4 [6]. In addition, we have collected the needs from different sources:

- Mainly, the periodic meetings with the development teams
- Messaging channels (e.g., Teams)

- The EMERALD WP1 mailing list (WP1 is in charge of the integration)
- The issue mechanism in GitLab

The inputs and outputs of this process can be summarised as follows:

- Inputs:
 - Requirements
 - Feedback from pilots' validation
 - EMERALD Architecture
 - External service requirements
- Outputs:
 - Summary of the activities performed in the last cycle
 - Activities to be performed in the next cycle
 - Activities backlog
 - Issues linked to the requirements.

2.3.2 Code

During the coding process, the integration of the different components is coded in a container orchestration code (aka choreography). In addition, code for the environment and side-services can be developed when needed. Once the development team has packaged or updated a component, a collaboration cycle with the DevOps team starts. During this cycle, the DevOps team supports the development team in different areas:

- **Repository creation:** The DevOps team creates a repository for each component in the EMERALD GitLab repository and provides the necessary access to the development team to manage their own component group.
- **Image creation:** The DevOps team provides support to the development team in the creation and publishing of the *Docker* image (or images) for the component. On the one hand, providing the EMERALD Artifactory repository (implemented in *Artifactory*³) to store the *Docker* image and, on the other hand, providing support in the creation of the *Dockerfile* using CI/CD GitLab components.
- **Orchestration creation:** A *Kustomize*⁴-based orchestration has been created to help the development teams to create the orchestration code. The baseline orchestration has been created in a modular way to allow the development teams to add and test their components easily.
- **Debugging support:** Different resources are provided by the DevOps team to help the development teams to debug their components. We started with the logging and the console access to the *Kubernetes* cluster. Later, based on the feedback from the development teams, we added more debugging resources to the development teams such as the *Keycloak* testing environment, or the local testing environment based on *Docker Compose*.
- **Update Support:** Guidelines and automatic update mechanisms are provided to the development teams to help them update their components in the CaaS Framework. The guidelines are provided in the EMERALD GitLab repository, and the automatic update mechanisms are implemented with *Renovate*⁵ and GitLab CI/CD.
- **Adding side-services:** The DevOps team helps with the creation of side-services that are needed to support the integration of the components. For example, adding administration frontends for databases to help the development teams in the analysis

³ <https://jfrog.com/artifactory/>

⁴ <https://kustomize.io>

⁵ <https://github.com/renovatebot/renovate>

of the data contained. On the other side, the development teams are responsible of applying the guidelines provided by the DevOps team to integrate and update their components in the CaaS Framework. The main outcome of this process is a *Kustomize* component for each of the components.

The inputs and outputs of this process can be summarised as:

- Inputs:
 - Components from the different development teams in a package repository (in EMERALD, implemented in *Artifactory*)
- Outputs:
 - Choreography and configuration code for the integration environment.
 - Infrastructure as code (IaC) for the environments.

2.3.3 Build

The building process creates a merged manifest for a specific *Kustomize* overlay. On the one hand, it can be done manually – by the DevOps or the development team – to generate a combined manifest to deploy the CaaS Framework. On the other hand, it can be done automatically as part of the CI/CD pipelines.

The inputs and outputs of this process can be summarised as:

- Inputs:
 - Packages from the components
 - Choreography code and configuration for the integration environment
 - Infrastructure as code (IaC) for the environments
- Outputs:
 - Integration environment
 - CaaS Framework in the integration environment.

2.3.4 Test

The test process covers several activities, such as the update of the candidate production environment based on the development environment, the creation of the integration tests, and the application of the integration tests over the candidate production environment.

First, the deployment of the candidate production environment is done automatically, together with the build process, in a CI/CD pipeline. Then, for the creation of the integration tests, three steps are performed:

- First, the DevOps team creates the integration tests that are needed to verify the functionality of the evidence collector components. These components are not deployed as part of the CaaS Framework services. Instead, they are deployed on the resources to be certified, in order to collect evidence of their compliance. Therefore, we need this mechanism to be able to verify the compatibility of the collectors with the CaaS Framework.
- Second, the DevOps team creates the integration tests covering the EMERALD workflows, as defined in D4.2 [20].
- Third, the DevOps team implements persistence tests to verify that the data in the CaaS Framework is not lost during the deployment of new versions of the CaaS Framework.

Finally, the integration tests are to be applied over the candidate production environment. The DevOps team provides the necessary resources to deploy the candidate production environment and to run the integration tests. The objective of the tests is to provide some

confidence on the correctness of the integration of the components before deploying them into the intended pilots.

The inputs and outputs of this process can be summarised as:

- Inputs:
 - CaaS Framework in the integration environment
 - List of collector components and their usage procedures
 - Main workflows defined as part of the D4.2 [20]
- Outputs:
 - Integration tests
 - Integration test results.

2.3.5 Release

During the planned release milestones, as well as when other releases require it, we will create a formal release of the CaaS Framework. There are two kinds of releases:

- Milestones releases: These are planned and will take place at M18, M30 and M34.
- Minor and fix releases: These may take place at any point based on the evolution of the components.

On the planned release milestones, the DevOps team creates a release commit over the CaaS Framework repository. This commit triggers a Release CI/CD pipeline that creates the candidate production environment and runs the integration tests over it. The results of the integration tests are used to decide if the release is successful or not. The EMERALD project applies semantic versioning [31] to the releases of the CaaS Framework. The planned releases apply a mayor upgrade (e.g., v1.0.0 to v2.0.0) to the version of the CaaS Framework.

Other releases are triggered by the development teams when they update their components. In this case, we use an automatic approach to introduce the new version of the component in the CaaS Framework. The automatic approach is based on the *Renovate* tool. This tool scans periodically (every week) changes in the images used in the manifests that compose the CaaS Framework. When a new version of the image is detected, it creates a merge request with the new version of the component. This merge request is automatically tested against the integration tests. If the integration tests are successful, the merge request is automatically merged into the CaaS Framework repository and a patch upgrade (e.g., v1.0.0 to v1.0.1) is applied to the version of the CaaS Framework.

The inputs and outputs of this process can be summarised as:

- Inputs:
 - Release request
 - Integration tests
- Outputs:
 - Integration test results
 - CaaS Framework formal release.

2.3.6 Deploy

This process deploys the formal release of the CaaS Framework in the production environments. We support two types of deployment:

- SaaS (Software as a Service): The CaaS Framework is deployed in the EMERALD infrastructure and the pilots consume it as a service. In this case, the deployment is done automatically as part of the Release CI/CD pipeline.
- On-premises: The CaaS Framework is deployed in a pilot infrastructure and the pilot consumes it as an on-premises configuration. In this case, the deployment is done manually.

The inputs and outputs of this process can be summarised as:

- Inputs:
 - CaaS Framework formal release
- Outputs:
 - Production environment as SaaS
 - Pilot environments, if required.

2.3.7 Operate

Besides the regular operation activities that are performed in the production environment, as for example:

- Capacity management
- Performance management
- Availability management
- Incident management
- Backup management

We also include in this stage the validation process, which is not performed by the DevOps team, but by the pilots and the development teams. Please note that the DevOps team is responsible for the validation of the CaaS Framework.

Two main aspects are covered in the Operate process:

- Gathering the feedback from the pilots and the development teams.
- Processing the feedback to improve the CaaS Framework.

Feedback is gathered through the EMERALD project communication channels. Mainly the periodic meetings with the development teams and the pilots, and the messaging channels (e.g., Teams). Feedback is processed by the DevOps team and the development teams and transformed into pending tasks in the backlog of the DevOps team. These tasks are prioritised and at given points they become into issues in the GitLab repository. The issues are then planned in the next iteration of the DevOps by the development teams.

The inputs and outputs of this process can be summarised as:

- Inputs:
 - Tagged CaaS Framework in the production environment (and pilot environments, if required)
- Outputs:
 - Feedback from pilots' validation.

2.3.8 Monitor

The monitoring of the integration environment is adapted in this stage to cover the production and pilots' environments. Based on the feedback from the pilots' monitoring, mechanisms are created to check the behaviour of the framework in the long term.

The inputs and outputs of this process can be summarised as:

- Inputs:
 - Tagged CaaS Framework in the production and pilot environments, if required
- Outputs:
 - Monitoring procedures
 - Monitoring results.

2.4 Lifecycle

There are several applicable lifecycles in software development [22], [23], but considering the following characteristics of the EMERALD project:

- There are different components with different set of requirement, different teams, and different agendas.
- There are fixed milestones (see *APPENDIX B: Project Milestones from the DoA*) at project level that should be achieved.

We decided to apply an iterative process, as stated in the DevOps lifecycle (see Figure 1). The DevOps iterations are continuous and are performed whenever the DevOps team receives integration and release requests from the development teams. Besides those requests, the DevOps team also performs iterations every two weeks, focussing on the environment's setup, integration tests, monitoring mechanisms, and monitoring results.

We use three mechanisms to document the tasks to be performed (e.g., implement monitoring service in k8sv) in the context of the DevOps activities: tasks, issues, and merge requests.

- **Tasks** are used in the context of the periodic meetings with the development teams and the pilots to document the tasks that are going to be performed in the next iteration. They are also used to document the tasks that will be performed afterwards, this is what we call the backlog of tasks.
- **Issues**⁶ are used as the primary mechanism for documenting tasks that involve some effort on the part of the DevOps team. Every task is documented in an issue inside the affected repos under the DevOps group in the project GitLab repository (see Figure 2).
- **Merge requests**⁷ are used change the code in order to support the completion of the issues created (see Figure 3).

⁶ <https://docs.gitlab.com/ee/user/project/issues/>

⁷ https://docs.gitlab.com/ee/user/project/merge_requests/

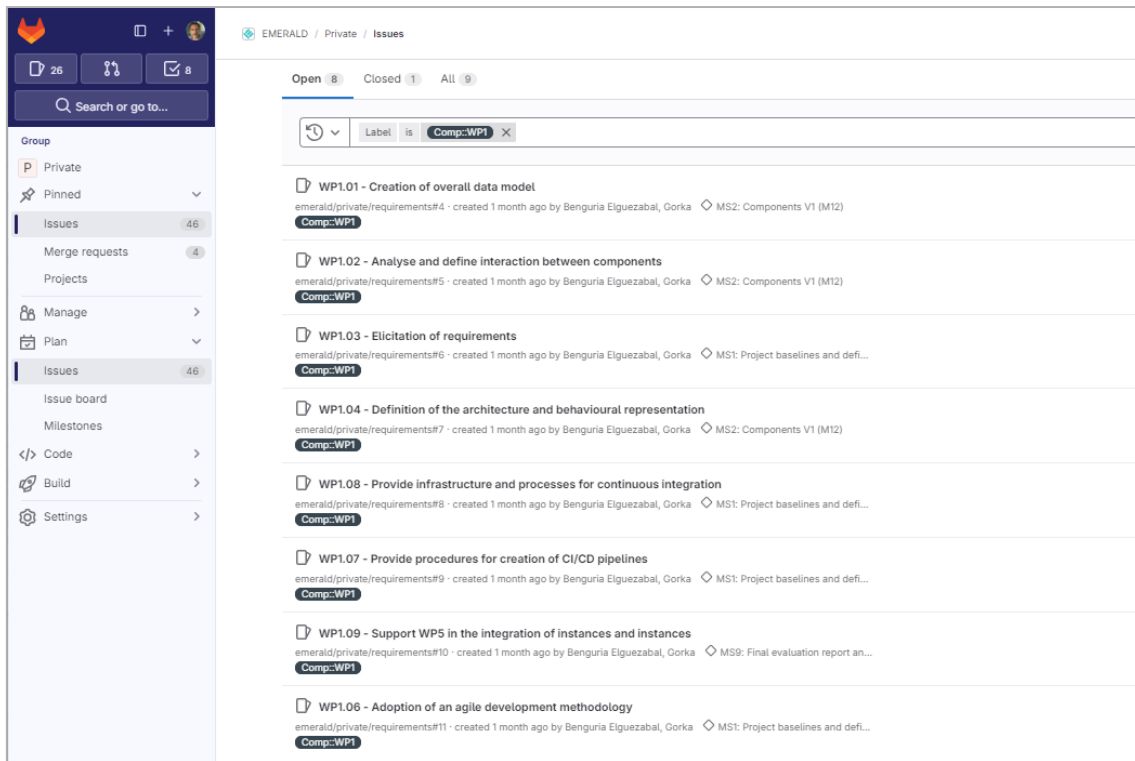


Figure 2. List of Issues related to Concept & Methodology in EMERALD

In the case of issues and merge requests, they must be related with some requirement(s) in order to keep track of its implementation in the DevOps activities. Issues are linked with the requirements using the “linked requirements” mechanism provided by GitLab. In the case of merge requests, if they have not been created from an issue, they will be related with an issue in the description and that issue will be related with some requirement(s).

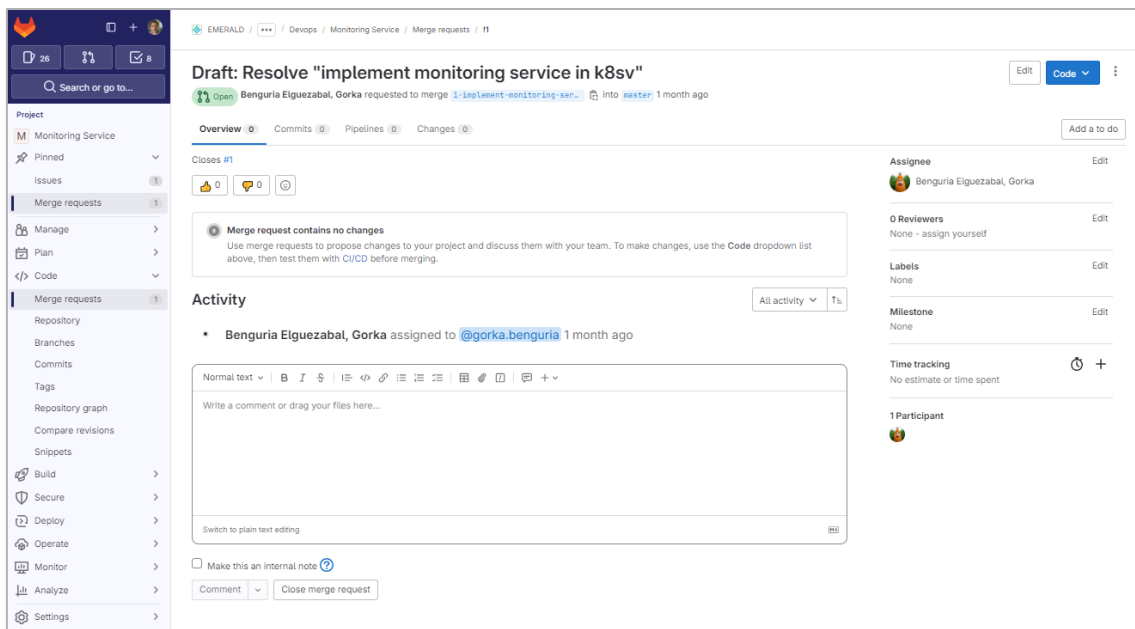


Figure 3. A Merge Request related to Concept & Methodology in EMERALD

Tasks are managed in an agile approach, with some flexibility. For example, if there is an urgent need to quickly update components at the request of the development teams they will be done

directly. Regular tasks enter the DevOps lifecycle as requests in the task backlog (i.e., as issues in the DevOps repository).

At the end of each iteration (i.e., bi-weekly), we perform an internal meeting in WP1 to:

- Review the status of completion of the tasks worked out during the iteration.
- Choose the new set of tasks to be carried out during the next iteration.

The status of the DevOps tasks is visible at the DevOps group level in GitLab and is recorded in the minutes of the periodic WP1 meetings with the development teams and the pilots.

The execution of each task may involve one or more planned processes (except for the monitoring process that is automated and continuous), such as Code, Build, Test, Release, Deploy, and Operate. These processes are not expected to be carried out in every task, i.e., depending on the nature of the task, it may involve all or some of them, and the effort in each task varies.

3 CI/CD Strategy

This section provides details on the strategies and technical approach applied during the development of the first version of the CaaS Framework. Besides, we also outline some strategies that will be matured during the next stages of the project, such as those that will support the upcoming releases of the CaaS Framework, as well as the deployment of the CaaS Framework in the pilots. The section is divided into two main parts: CI Strategy and CD Strategy.

The description contains much information in common with D1.5 [1] with the final aim of providing a self-contained section that facilitates the reader's understanding.

3.1 CI Strategy

For the integration of the outcomes of the development teams in the CaaS Framework, we have applied the following technological approaches:

- Components packaged as containers
- Environment defined with Infrastructure as Code (IaC)
- Integration guidelines
- CI/CD Components
- Component based *Kustomize*
- Manual deployment support
- *Rancher* for debugging support
- Local environment for testing
- Progressive verification
- Integration automation.

3.1.1 Container-based

Container technology has proved to be a very good approach to aggregate components from different teams. Besides, if used appropriately, it also provides de facto scalability and resilience when we use container orchestration technologies, such as *Kubernetes*⁸ or *Docker Swarm*⁹. In addition, the usage of the container technology promotes decoupling from the architecture which provides some benefits over monolithic architectures [24], [25].

EMERALD prioritises container images as the default packaging technology for its components. In case some components cannot be deployed as containers, IaC and service approaches are prioritized as backup strategies.

As a container technology, we have used the *Docker* ecosystem to build and share images. For image building we support both *Docker* and *Docker Compose*. A component of the EMERALD architecture may include one or more *dockerfiles*¹⁰ to build the images that are used to deploy the CaaS Framework. In some cases, the building process may require some orchestration. In those cases, *Docker Compose* or *Custom Scripting* is used as well.

To support both scenarios as part of the DevOps activities, we have provided resources and support for the automation of building such images. We have used different technologies including:

- **Docker shared runners** that support *Docker in Docker* (dind)¹¹ technology

⁸ <https://kubernetes.io/>

⁹ <https://docs.docker.com/engine/swarm/>

¹⁰ <https://docs.docker.com/build/concepts/dockerfile/>

¹¹ <https://www.docker.com/resources/docker-in-docker-containerized-ci-workflows-dockercon-2023/>

- **Kubernetes group runners**
- **Docker machine shared runners** that support dind technology

As an IaC technology, *OpenTofu*¹² and *Ansible*¹³ have been favoured, since both technologies are open-source and facilitate knowledge sharing and latter distribution of the resources.

Finally, in case there are some components that cannot be physically deployed in production and must be consumed as a service, the OpenAPI¹⁴ Specification is promoted.

Regarding the strategy with respect to the packaging on behalf of the DevOps team, it is planned to work as follows:

- Provide example packaging approaches starting from a *Docker* or a *Docker Compose* specification. These examples are specific to the GitLab framework used in the EMERALD project and include:
 - `.gitlab-ci.yml` for *Docker* and `gitlab-ci.yml` for *Docker Compose*.
 - Example of a dockerfile.
 - Example of a docker compose file.
 - Readme file with indications on how to integrate the component.
- Each example includes guides on how to integrate it on the components that are stored in the GitLab framework used in the EMERALD project.
- When necessary, WP1 has provided support in the integration of the gitlab-ci, and in the development of *Docker* and *Docker Compose*. This has been managed through a merge request from the interested party, which has been documented in an issue related to a project requirement.

3.1.2 Environments with IaC

The environments that support the CaaS Framework integration and validation were developed following an IaC approach with state of the practice open-source tools.

The integration environment has been developed following this approach. For that, a project has been created in the private GitLab of EMERALD (`emerald/private/devops/opentofu-k8sv`). This project creates a three-node *Kubernetes* cluster over *vSphere* platform and includes instructions to replicate the deployment on another *vSphere* platform if necessary. In addition, it uses a reusable set of *Ansible* playbooks to configure the EMERALD *Kubernetes* that have been applied in this case and could also be applied on other nodes with little or no customization.

The IaC is configurable through the modification of two templated yaml files:

- **Base *OpenTofu* hosts:** (`/blob/master/base_opentofu_hosts.yaml.erb`), that controls the initial creation of the virtual machines and their configuration by *Ansible*.
- ***Ansible* host:** (`/blob/master/Ansible_hosts.yaml.erb`), that configures the machines using some *Ansible*-playbooks.

The same approach was followed for the instantiation of other environment and resources, so that we can replicate them in case we need to do so in latter stages of the project.

With the project already started (M7), the integration environment was moved from *vSphere* to *OpenStack*. In this migration we forked the IaC (`emerald/private/devops/opentofu-k8sv`) into a new project (`emerald/private/devops/opentofu-k8so`). In this project we added the

¹² <https://opentofu.org/>

¹³ <https://docs.ansible.com/>

¹⁴ <https://www.openapis.org/>

necessary changes to deploy the CaaS Framework in *OpenStack*. The new project is based on the same principles as the previous one, but it uses *OpenStack* as the infrastructure provider.

In the same way that any other activity in the DevOps team, the IaC development was documented in an issue related to an EMERALD project requirement and implemented through a merge request.

3.1.3 Integration guidelines

As part of the integration support activities, the DevOps team has developed a set of guidelines to help in the integration of the components in the CaaS Framework. The guidelines audience is currently the EMERALD development team.

These guidelines, that have been published in the EMERALD GitLab repository public area (<https://git.code.tecnalia.dev/emerald/public/contribute>), will evolve as the project advances and as we improve the integration process. Currently the Guidelines include:

- **Enrol:** How to join to the repository for contributions.
- **Component Development:** Guidelines for developing a new component for the CaaS Framework.
- **Component Integration:** Guidelines for integrating a new component into the CaaS Framework.
- **Component Validation:** Guidelines for validating a new component for the CaaS Framework.
- **Keycloak Integration:** Guidelines for integrating Keycloak into the components of the CaaS Framework.
- **Component Update:** Guidelines for updating a component of the CaaS Framework.
- **Component Troubleshooting:** Guidelines for troubleshooting a component of the CaaS Framework.
- **Semantic Versioning with GitLab CI/CD:** Guidelines for versioning the components of the CaaS Framework.
- **Component Publish with GitLab CI/CD:** Guidelines for setting up the CI/CD pipeline for publishing a component image in GitLab.

In addition, a training session was organized with the development teams to explain these guidelines and how to use them. The session was held during the General Assembly of the EMERALD project in Barcelona, on 23rd and 24th of October 2024.

As mentioned above, the guidelines evolve as the project progresses. Significant changes to the guidelines are presented at regular meetings with the development teams and pilot owners.

3.1.4 CI/CD Components

In order to help the development teams and the DevOps team to automate the CI/CD pipelines, we have introduced CI/CD components into the EMERALD project. The CI/CD components are a set of reusable components that can be used to automate the CI/CD pipelines.

For the provision of these components, we have reused some GitLab CI/CD components already provided by the community. In particular, we have used components provided by "to be continuous"¹⁵ as baseline. Over this base we have introduced the necessary changes to make them work in the project infrastructure. The CI/CD components are accessible at <https://git.code.tecnalia.dev/explore/catalog>.

¹⁵ <https://to-be-continuous.gitlab.io/>

For the development teams we have introduced several components:

- **Docker** (<https://git.code.tecnalia.dev/explore/catalog/smartdatalab/public/ci-cd-components/docker>): This component is used to build the *Docker* images and publish them in an *Artifactory*. Additionally, it also allows to perform security checks based on *Trivy*¹⁶.
- **Semantic Release** (<https://git.code.tecnalia.dev/explore/catalog/smartdatalab/public/ci-cd-components/semantic-release>): This component is used to create releases following the semantic versioning approach. It parses commit messages and based on their content it modifies the version of the component creating a tag and a release. Additionally, it can also be configured to perform additional tasks, such as creating the changelog or modifying the version within the code.
- **Gitleaks** (<https://git.code.tecnalia.dev/explore/catalog/smartdatalab/public/ci-cd-components/gitleaks>): This component is used to verify that the code pushed into the master branch does not contains any password in the code.

For the DevOps team, apart from the previous components, we have also introduced the following components:

- **Kubernetes** (<https://git.code.tecnalia.dev/explore/catalog/smartdatalab/public/ci-cd-components/kubernetes>): This component is used to deploy the CaaS Framework in the integration environment.
- **Renovate** (<https://git.code.tecnalia.dev/explore/catalog/smartdatalab/public/ci-cd-components/renovate>): This component is used to configure projects and scan their dependencies. For each dependency, it checks if there is a new version available and if this is the case, it creates a merge request with the new version of the dependency. This component is used to keep the CaaS Framework up to date with the latest versions of the components.

Additionally, for some components we have added language specific components:

- **Maven** (<https://git.code.tecnalia.dev/explore/catalog/smartdatalab/public/ci-cd-components/maven>): This component is used to build and test *Maven* projects. It can also be used to publish them in an *Artifactory*, but in EMERALD we are not using this feature as we are packaging the components as *Docker* images.
- **SonarQube** (<https://git.code.tecnalia.dev/explore/catalog/smartdatalab/public/ci-cd-components/sonar>): This component was introduced as a proof of concept for one of the components.

Together with the CI/CD components, we have added some examples on how to include them in the CI/CD pipelines. The examples are pointed in the description of some of the CI/CD components¹⁷. Most relevant examples for the EMERALD project are:

- **CI/CD pipeline for Docker** (<https://git.code.tecnalia.dev/smartdatalab/libraries/ci-cd-samples/docker>)¹⁸: This example uses the *Docker* component to build and publish the *Docker* image.

¹⁶ <https://aquasec.com/products/trivy/>

¹⁷ Most of them are stored at <https://git.code.tecnalia.dev/smartdatalab/libraries/ci-cd-samples> for internal access. But *Listing 1. CI/CD for the RCM* and *Listing 2. CI/CD for side-service* are some examples derived from them that are detailed in *APPENDIX C: Integration files*.

¹⁸ This is an internal link, which is also detailed in *APPENDIX C: Integration files*

- **CI/CD pipeline for Semantic Release**
(<https://git.code.tecnalia.dev/smartdatalab/libraries/ci-cd-samples/semantic-release/>¹⁹): This example uses the Semantic Release component to create the release over a dockerfile project.

These CI/CD components are being used in the components of the CaaS Framework. For example, the RCM component uses several of them in their CI/CD pipelines, such as the *Docker* component to build the Docker image and publish it in the *Artifactory*, the *Semantic Release* component to create the release over the dockerfile project, and the *Gitleaks* component to verify that the code pushed into the master branch does not contains any password in the code. Besides, the RCM also uses the *Maven* component to build and test the Maven projects. This usage is visible in the private part of the EMERALD GitLab repository (emerald/private/components/rcm); therefore, we include below the content of one of the .gitlab-ci.yml files as an example of the usage of the CI/CD components.

The .gitlab-ci.yml for the RCM component is included in Listing 1. In the code under the include element the components used are added as component elements. Each component includes a reference to the components and the inputs to modify their default behaviour. In the code below we can see *Docker*, *Maven*, *Semantic release* and *Gitleaks* components.

```
include:
  - component: git.code.tecnalia.dev/smartdatalab/public/ci-cd-
    components/docker/gitlab-ci-docker@master
  inputs:
    snapshot-image: $CI_REGISTRY_IMAGE/snapshot:$CI_COMMIT_REF_SLUG
    release-image: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_NAME
    hadolint-job-tags: ["docker"]
    kaniko-build-job-tags: ["docker"]
    dind-build-job-tags: ["docker"]
    buildah-build-job-tags: ["docker"]
    healthcheck-job-tags: ["docker"]
    docker-trivy-job-tags: ["docker"]
    docker-sbom-job-tags: ["docker"]
    docker-publish-job-tags: ["docker"]
    metadata: >-
      --label org.opencontainers.image.url=${CI_PROJECT_URL}
      --label org.opencontainers.image.source=${CI_PROJECT_URL}
      --label org.opencontainers.image.title=${CI_PROJECT_PATH}
      --label org.opencontainers.image.ref.name=${CI_COMMIT_REF_NAME}
      --label org.opencontainers.image.revision=${CI_COMMIT_SHA}
      --label org.opencontainers.image.created=${CI_JOB_STARTED_AT}
      --label com.jfrog.artifactory.retention.maxCount=4
    prod-publish-strategy: auto
    hadolint-disabled: true
    healthcheck-disabled: true
    trivy-disabled: true
    sbom-disabled: true
    build-tool: dind
  - component: "git.code.tecnalia.dev/smartdatalab/public/ci-cd-
    components/maven/gitlab-ci-maven@master"
  inputs:
    image: maven:3.8.1-openjdk-17-slim
    mvn-build-job-tags: ["docker"]
    mvn-dependency-check-job-tags: ["docker"]
    mvn-no-snapshot-deps-job-tags: ["docker"]
    mvn-sbom-job-tags: ["docker"]
    mvn-sonar-job-tags: ["docker"]
    build-args: org.jacoco:jacoco-maven-plugin:prepare-agent package
    org.jacoco:jacoco-maven-plugin:report
    sonar-base-args: clean test sonar:sonar -Dsonar.links.homepage=${CI_PROJECT_URL}
    -Dsonar.links.ci=${CI_PROJECT_URL}/-/pipelines -Dsonar.links.issue=${CI_PROJECT_URL}/-/
    /issues ${SONAR_TOKEN:+-Dsonar.login=${SONAR_TOKEN}}
    mvn-semrel-release-disabled: "false"
  - component: "git.code.tecnalia.dev/smartdatalab/public/ci-cd-components/semantic-
    release/gitlab-ci-semrel@master"
```

¹⁹ This is an internal link, which is also detailed in *APPENDIX C: Integration files*

```
inputs:
  auto-release-enabled: true
  release-disabled: false
  semantic-release-job-tags: ["docker"]
  branches-ref: "/^(master|main)$/"/>
  image: timbru31/java-node:17-jdk-iron
- component: "git.code.tecnalia.dev/smartdatalab/public/ci-cd-
components/gitleaks/gitlab-ci-gitleaks@master"
  inputs:
    gitleaks-job-tags: ["docker"]

variables:
  GIT_STRATEGY: clone
  CI_REGISTRY_IMAGE: emerald-docker-dev-local.artifact.tecnalia.dev/rcm/backend
```

Listing 1. CI/CD for the RCM component

These components are also being used by the DevOps team. For example, we use them in the side-service and the CaaS Framework *Kustomize* composition. In this composition we use the *Kubernetes* component to deploy to the Kubernetes environment. We also use the *Semantic Release* component to create the release over what is deployed. This usage is visible only in the private part of the EMERALD GitLab repository (for example, `emerald/private/devops/Side-services`), therefore we include below the content of one of the `.gitlab-ci.yml` files as an example of the usage of the CI/CD components.

The `.gitlab-ci.yml` for side-services is included in Listing 2. In the code under the `include` element the components used are added in component elements. Each component includes a reference to the components and the inputs to modify their default behaviour. In the code bellow (Listing 2) we can see *Kubernetes* and *Semantic Release* components.

```
include:
- component: git.code.tecnalia.dev/smartdatalab/public/ci-cd-
components/kubernetes/gitlab-ci-k8s@master
  inputs:
    kustomize-enabled: true # latter we will use it
    score-disabled: true
    prod-space: "emerald-ext"
    prod-deploy-strategy: auto
    k8s-score-job-tags: ["docker"]
    k8s-review-job-tags: ["docker"]
    k8s-cleanup-review-job-tags: ["docker"]
    k8s-integ-job-tags: ["docker"]
    k8s-staging-job-tags: ["docker"]
    k8s-prod-job-tags: ["docker"]
    prod-url: https://k8so.emerald.digital.tecnalia.dev/k8s/clusters/local
- component: "git.code.tecnalia.dev/smartdatalab/public/ci-cd-components/semantic-
release/gitlab-ci-semrel@master"
  inputs:
    auto-release-enabled: true
    release-disabled: false
    semantic-release-job-tags: ["docker"]
    branches-ref: "/^(master|main)$/"/>

Variables:
  KUBE_CONTEXT: emerald/private/devops/gitlab-agent-k8so:emerald-ext # comment when
gitlab agent for k8s stops working, and define KUBECONFIG in project variable with
values taken from rancher
  GIT_STRATEGY: clone
```

Listing 2. CI/CD for side-service

3.1.5 Component-based Kustomize

Under the hood, the deployment in Kubernetes is based on manifests. This is manageable for small deployments, but as the number of components increases, it becomes harder to manage.

Besides, it is not flexible enough to support the different environments that we need to deploy. For that reason, we have used *Kustomize*²⁰ as the main tool to manage the manifests.

Kustomize allows to define components and overlays that fit the needs of the project. The components are used to define the basic configuration of the EMERALD components, while the overlays are used to adjust the configuration of the components to the specific environments. The components can be seen in the public repository of the EMERALD project (<https://git.code.tecnalia.dev/emerald/public/caas-framework/-/tree/master/components>). In that folder we mix EMERALD components with other components required by the CaaS Framework or required for testing it, such as:

- **artifactory-secrets:** This component configures the EMERALD *Artifactory* secret so that the *Kubernetes* cluster can download the images from it.
- **keycloak-secrets:** This component configures the EMERALD *Keycloak* secret so that the EMERALD components can use that secret to inject their *Keycloak* configuration.
- **keycloak-loader:** This component is under deprecation. It was used to load a merged *Keycloak* configuration into the EMERALD *Keycloak*. But currently, we are moving to a component-based approach, where each component loads its own configuration into the EMERALD *Keycloak*.
- **keycloak-test:** this component is used in case the developers want to test their *Keycloak* configuration in a testing *Keycloak*. It is not used in the production environment.

The overlays are placed in the root folder of the repository (<https://git.code.tecnalia.dev/emerald/public/caas-framework/-/tree/master>). Currently, at M18, we only have the integration overlay (<https://git.code.tecnalia.dev/emerald/public/caas-framework/-/tree/master/integration>). In the next releases we will include the production overlay in order to systematize the creation of new releases of the CaaS Framework.

The usage of *Kustomize* facilitates the deployment of the CaaS Framework. For example, to deploy the whole development environment, providing the necessary access rights, the DevOps team only needs to run the following commands:

```
```bash
kubectl config use-context devops-kubeconfig
kustomize build integration | kubectl apply -f -
```
```

On the developer side, as explained in the integration guidelines, they can also destroy and deploy their components in the development environment without affecting the other components. To do so, they need to run the following commands:

```
```bash
kubectl config use-context developer-kubeconfig
kustomize build components/<component_name> | kubectl delete -f -
kustomize build components/<component_name> | kubectl apply -f -
```
```

This allows the developers to quickly test their components in the development environment without the need to wait for the CI/CD pipeline to finish.

3.1.6 Manual deployment support

Once a component is ready to be integrated into the CaaS Framework, it is very common that minor changes and tests are required in the component and in the way in which it is integrated into the CaaS Framework. Performing these changes through the CI/CD pipeline may take a long

²⁰ <https://kustomize.io>

time, as it requires the developer to wait for the CI/CD pipeline to finish. This involves time, patience and concentration from the developer, and computing resources from the EMERALD project that might demotivate the developer, which is not a desirable situation.

Therefore, in EMERALD, when working against the CaaS Framework development environment, we always try to provide a simple, fast and efficient way to manually deploy the changes over a component into the CaaS Framework:

- Developers can build locally the images and push them to the EMERALD Artifactory. This is done through the CI/CD pipeline, but it can also be done manually.
- Developers can deploy the component in the CaaS Framework development environment. This is done through the CI/CD pipeline, but it can also be done manually.

Besides, developers are provided with CLI-based access to the CaaS Framework development environment. This allows them to:

- quickly access logs
- copy files
- execute commands in their components
- or even create temporary containers to test their pods

Bellow, we include an example of the commands that any developer can use in order to update a component in the CaaS Framework development environment and interact with it. In this example we are trying to understand a problem in the RCM component that is running in the CaaS Framework development environment. We need to test something in the RCM component, so we run the following commands (Listing 3):

```
```bash
echo "introduce a minor test change in the code"
cd ~\git\emerald\private\components\rcm\caas-framework
docker build -t emerald-docker-dev-local.artifact.tecnalia.dev/rcm/backend:latest .
docker push emerald-docker-dev-local.artifact.tecnalia.dev/rcm/backend:latest
cd ~\git\emerald\private\components\rcm\backend
kubectl config use-context developer-kubeconfig
kustomize build components/rcm | kubectl delete -f -
kustomize build components/rcm | kubectl apply -f -
kubectl logs -f $(kubectl get pods -l app=rcm -o jsonpath='{.items[0].metadata.name}')
kubectl cp $(kubectl get pods -l app=rcm -o jsonpath='{.items[0].metadata.name}'):/tmp/rcm.log .
kubectl exec -it $(kubectl get pods -l app=rcm -o jsonpath='{.items[0].metadata.name}') -- cat /tmp/rcm.log
kubectl run --rm -it --image=ubuntu --restart=Never -- bash
curl -X POST -H "Content-Type: application/json" -d
'{"username":"admin","password":"admin"}' http://rcm:8080/rcm/api/v1/login
exit
echo "after evaluation the results we can continue with another test or implement a
formal fix"
```
```

Listing 3. Example commands to manually redeploy a component

In the code described above we can see how to:

- Build the component image and upload to the *Artifactory* (docker build and docker push)
- Remove the composition from the development environment (kustomize build and kubectl delete) and launch the composition that will reload the new component image from the *Artifactory* (kustomize build and kubectl apply)
- Check the logs of the component (kubectl logs)
- Copy a file from inside the component (kubectl cp)

- Execute a command into the component (`kubectl exec`)
- Check the RCM API internally (`curl`). To do so we create a temporal component in *Kubernetes* to check access to services from inside the cluster (`kubectl run --rm`)

3.1.7 Rancher for debugging support

Besides the manual deployment support, we also provide a *Rancher*-based web interface to manage the *Kubernetes* cluster. This is convenient for many reasons:

- Provides a user management interface
- Allows to quickly check the status of the deployments
- Provides a fast way to check the logs of the pods

User management is necessary in EMERALD because we try to simulate a realistic deployment scenario, and it is not realistic to have an admin access when deploying the CaaS Framework in the pilots. The *Rancher* user management has been used to create a highly constrain *emerald-developer* user, which has only write-access on the emerald-dev project and read-access to the Keycloak project for debugging purposes.

Besides, it is also very useful to have quick access to monitoring and debugging tools. It is very common in the DevOps team to be inquired about the status of some aspect of the CaaS Framework (e.g., the status of a component, the status of a deployment, or the status of a pod). Initiating the developer CLI requires time and sometimes it may be more complex than expected. In these cases, the *Rancher* interface provides a quick access to this information. The only things required are the URL of the *Rancher* interface (<https://k8so.emerald.digital.tecnalia.dev/>) and the user credentials.

3.1.8 Local environment for testing

A partial local environment based on *Docker Compose* has been created. This environment is not intended to be used in production, but it is useful for some development and debugging situations. It provides a faster mechanism to test interaction between components. This local environment has been used in two situations:

- To understand the configuration of the consul service required by the RCM component.
- To understand the configuration of *Keycloak* for several components.

The RCM local test requires several side-services to be able to run (i.e., a database and a consul service). These services require some initial configuration that must take place before the RCM modules are started. In this situation, the local environment was used to understand, sequence and test that configuration. Besides, the environment has shown to be useful not only to understand how to configure them but also to deploy them locally. Currently the RCM developers use the local environment when they develop the RCM component.

On the other hand, one of the challenges for the DevOps team is to understand the configuration of the *Keycloak* service required by the EMERALD components. One of the lessons learn from previous projects was that *Keycloak* configurations must be stored under version control. This enables to configure the *Keycloak* service before the EMERALD components are started. The local environment was used to understand, sequence and test that configuration.

3.1.9 Progressive Verification

The verification of the added and updated components of the CaaS Framework is an important aspect to ensure the secure evolution of the platform during the project. The verification is covered by a set of integration tests that are being automated.

Verification activities focus on multiple aspects:

- Establish the means to check the health of the components
- Verify the compatibility of the evidence collector components
- Define more complex integration tests based on EMERALD workflows

In the first stage of the project, as components are added, procedures should be implemented to check their health. These mechanisms are used during the integration tests to be performed after the update of each component, as well as during continuous monitoring.

As the project advances, additional integration tests are added, based on requests from the developers and feedback from the pilots.

The strategy in EMERALD is to document verification-focused activities as issues linked to the requirements. Implementing means to check the health of a component may require implementing parts in the choreography and parts in the component itself. More complex integration may require implementing specific components to generate the activity required to verify such complex integration scenarios.

3.1.10 Automation

DevOps activities focus on the automation of all the activities related to the evaluation of components as they are updated by the developers. The main focuses of automation are:

- **Update the integration platform** as the developers update the components.
- **Run the integration test** as the platform is updated.
- **Update the monitoring mechanism** to measure the health of the CaaS Framework in the long term.

The strategy is to use *GitLab Agent*²¹ for Kubernetes, implemented in the DevOps repository inside the *GitLab* repository of EMERALD (emerald/private/devops/gitlab-agent-k8so). It monitors the CaaS Framework repository, and every change detected there is translated into the aimed environments, which allows to deploy new component versions directly, without integration testing. This is done to speed up the feedback to the development team. Integration tests are started at a later stage, using a *GitLab runner*.

The monitoring mechanism is updated following the same approach as the CaaS Framework, i.e., we use the *GitLab Agent* for Kubernetes for this purpose as well.

3.2 CD Strategy

For the deployment of the CaaS Framework to be evaluated by the project and the pilots, the following technological approaches are applied:

- Releases
- Public Assets Release
- Keycloak Configuration
- Demo pilot
- Documentation
- Environment defined with IaC
- Deployment automation

²¹ <https://docs.gitlab.com/user/clusters/agent/install/>

3.2.1 Releases

The project follows a versioning system with three mayor releases:

- v1.0.0 - First release of the EMERALD components in month 18
- v2.0.0 - Second release of EMERALD components in month 30
- v3.0.0 - Final release of EMERALD integrated audit suite in month 34

Additional releases are expected between those mayor releases, as the project advances and the CaaS Framework is validated. Versions v1.x.x have been created during the first iterations of the project before the month 12. Versions v3.x.x are also expected based on the feedback of the last validation activities.

The strategy for the releases starts with a petition from an EMERALD partner (see Figure 4). Basically, the release consists of moving the content from the integration branch to the production branch. To perform this process an issue is created describing the request and the purpose. From that issue a new draft merge request is created over the production branch. That creates a new working branch, where we move the integration version that we want to move to production. The integration version to be deployed should have been successfully verified with the integration tests, otherwise we notify the risk before proceeding. After that, we change the draft status on the merge request, which enables the merge action over the production branch. We perform the merging that updates the production version. Finally, we communicate the change to the EMERALD project.

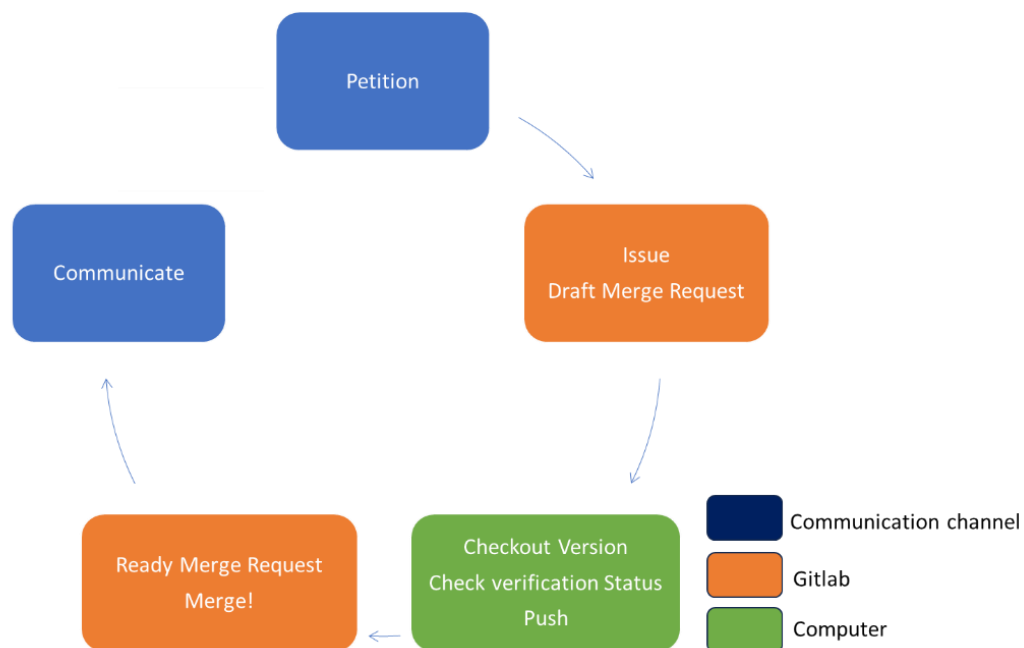


Figure 4. A Merge Request mechanism to produce a new release in EMERALD

3.2.2 Public Assets Release

For the public release of the CaaS Framework (<https://git.code.tecnalia.dev/emerald/public>), we have defined the following strategy:

- A common structure for the components has been defined that is replicated in the private and public repositories.
- An automated procedure has been implemented to promote the components from the private to the public repository.

- The CaaS Framework is located at the root of the public repository.
- A description of the public repository is included at the top of the public repository.

3.2.3 Keycloak configuration

The strategy with regards to the identity and access management is to assume it as an external service. We follow this approach because, in a real deployment, we could be facing a situation in which we are requested to use an existing identity and access management service in which we may not have administration rights.

So, the strategy for the *Keycloak* configuration management is reducing the administration requirements to the minimum. In this case, we will assume that in order to deploy the EMERALD CaaS framework we will only require a pre-existing realm and a user with rights to configure that realm.

Using that user, during the startup of the EMERALD framework, we upload the configuration of each component to the *Keycloak* service. This is done through a *Keycloak* loading job in each of the components. This supports the following objectives:

- Reduce the configuration requirements to the minimum.
- Allow the components to be deployed in a pre-existing *Keycloak* service.
- Facilitate the evolution of the *Keycloak* configuration of the components in an isolated way.

3.2.4 Demo pilot

One of the main goals of the EMERALD methodology, apart from the creation of the CaaS Framework, is to facilitate the validation of the CaaS Framework in the pilots. With the purpose of checking the deployment of a Demo pilot and its update, a separate demo environment will be created. So, we will create a repository for the demo pilot (emerald/private/devops/demo) that will contain the necessary elements for deployment:

- Configuration files for the demo pilot
- Secrets for the demo pilot
- Keycloak deployment for the demo pilot
- CaaS Framework deployment for the demo pilot.

3.2.5 Documentation

The documentation regarding the CaaS Framework is generated as part of the deployment to production. So, the focus of the documentation is on the deployment and configuration of the pilots.

We provide the installation and configuration instructions for the CaaS Framework in the README.md file of the public repository, which includes information about:

- Initial deployment of the CaaS Framework. Including details on how to configure the CaaS Framework to adjust to the specific needs of the pilots
- Update of the CaaS Framework
- Information about the feedback channels
- Information about the contribution mechanisms
- Information about the project roadmap
- General information about Authors, Acknowledgment and License
- Information about the project status

More contents will be added to the documentation in case it is necessary, for example to explain the usage of the evidence collector components in the pilots.

Currently the documentation is available in the public repository of the EMERALD project (<https://git.code.tecnalia.dev/emerald/public/caas-framework>). The documentation is included in the README.md file of the public repository.

Other documentation mechanisms may be added in the future to complement the installation and configuration information, such as the information generated using the pages feature of *GitLab*²².

3.2.6 Environments with IaC

The production environment is deployed using IaC, with the same approach used for the integration environment. Besides, the DevOps team will support the generation of additional environments, if needed. For example, environments on pilots' premises due to privacy or legal restrictions.

The strategy for the production environment IaC is similar to the strategy with the integration IaC. That is, as any other activity in the DevOps team, it is documented in an issue related to an EMERALD project requirement and implemented through a merge request.

3.2.7 Automation

The DevOps strategy works towards the automation of all activities related to the creation of releases and their deployment in the production environment. The main aspects of automation are focused on:

- **Deploying specific releases** to specific environments.
- **Updating the monitoring mechanism** to measure the long-term health of the CaaS Framework.

The automation strategy during deployment is similar to that of integration. We use *GitLab Agent for Kubernetes* to translate new releases on the production environment. For the pilot environments, we leave it up to the pilot owners to decide the procedure for updating their respective environments.

²² <https://docs.gitlab.com/user/project/pages/>

4 Conclusions

This document is the second and final version of the DevOps approach for EMERALD, where we have leveraged the baselines of the DevOps strategy presented in the previous version a year ago. In a year's time, we have put in practice the initial ideas to release the first version of the CaaS Framework, learnt some lessons and further develop the integration approach. The result of this process is this updated version of the report, containing a detailed description of the CI/CD process implemented.

The main guiding principle of this version is to provide the elements that help in the support of the transition of the EMERALD CaaS Framework to the pilots. This support will aim not only the initial deployment of the CaaS Framework, but also the evolution of the CaaS Framework during the project.

The methodology used customizes the commonly used DevOps lifecycle to the characteristics and constraints of the EMERALD project. This lifecycle consists of the following steps: Plan, Code, Build, Test, Release, Deploy, Operate, and Monitor. The goals of the defined methodology are to be release-based, manage feedback, manage components, keep traceability, manage the environments, and integrate as soon as possible. In this line, we have presented the main customised processes, as well as a tailored approach to iterate, so that we prioritise speed of integration over other elements.

In the CI/CD strategy part, we have described the technical approaches that we will implement to support the EMERALD project needs. In this sense, we will leverage some technologies and state of the practice DevOps resources, such as:

- Configuration management with IaC
- *GitLab* features with respect to:
 - Issues
 - Git workflows (branches and merge requests)
 - Automation with GitLab CI/CD Components
 - Documentation
- Releases with containers
- Container orchestration technologies

It is likely that during the course of the project some changes will be made to the DevOps methodology and the CI/CD Strategy, however the fundamental processes have already been defined in this deliverable and are in operation.

5 References

- [1] EMERALD Consortium, “D1.5 DevOps methodology and CI/CD strategy for EMERALD-v1,” 2024.
- [2] EMERALD Consortium, “Home page,” [Online]. Available: <https://www.emerald-he.eu/>. [Accessed April 2025].
- [3] EMERALD Consortium, “EMERALD - Annex 1 - Description of Action - GA 101120688,” 2022.
- [4] ISO, “ISO 16290:2013, Space systems — Definition of the Technology Readiness Levels (TRLs) and their criteria of assessment,” 2013.
- [5] EMERALD Consortium, “D1.3 EMERALD solution architecture-v1,” 2024.
- [6] EMERALD Consortium, “D1.4 EMERALD solution architecture - V2,” 2025.
- [7] CMMI Dev, “CMMI for Development, Version 1.3,” Software Engineering Institute (SEI), Ed., 2010.
- [8] International Organisation for Standardization (ISO/IEC), “ISO/IEC 15504-1:2004 Information Technology – Process Assessment – Part 1: Concepts and Vocabulary,” 2004.
- [9] AXELOS, “ITIL Foundation,” Stationery Office Books, Norwich, England, 2019.
- [10] J. A. V. M. K. Jayakody and a. W. M. J. I. Wijayanayake, “Process Improvement Framework for DevOps Adoption in Software Development,” in *2023 International Research Conference on Smart Computing and Systems Engineering (SCSE), IEEE, Jun. 2023*. doi: 10.1109/scse59836.2023.10214992, 2023.
- [11] I. Bucena and M. Kirikova, “Simplifying the DevOps Adoption Process,” in *BIR Workshops*, pp. 1–15, 2017.
- [12] R. d. Feijter, “Towards the adoption of DevOps in software product organizations: A Maturity Model Approach,” Master’s Thesis, 2017.
- [13] S. Badshah, A. A. Khan and B. Khan, “Towards Process Improvement in DevOps: A Systematic Literature Review,” in *Proceedings of the Evaluation and Assessment in Software Engineering, EASE ’20. ACM, Apr. 2020*. doi: 10.1145/3383219.3383280.
- [14] R. Amaro, R. Pereira and M. M. da Silva, “Capabilities and Practices in DevOps: A Multivocal Literature Review,” in *IEEE Trans. Softw. Eng.*, vol. 49, no. 2, pp. 883–901, Feb. 2023, doi: 10.1109/tse.2022.3166626.
- [15] M. Gasparaitė and S. Ragaišis, “Comparison of devops maturity models,” in *IVUS 2019. Proceedings of the International Conference on Information Technologies Kaunas, Lithuania, April 25, 2019, CEUR-WS. org, 2019*, pp. 65–69.
- [16] R. T. Yarlagadda, “DevOps and its practices,” in *Int. J. Creat. Res. Thoughts IJCRT ISSN*, pp. 2320–2882, 2021.
- [17] A. Colantoni, L. Berardinelli and M. Wimmer, “DevopsML: Towards modeling devops processes and platforms,” in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pp. 1–1, 2020.
- [18] R. Amaro, R. Pereira and M. M. da Silva, “DevOps Metrics and KPIs: A Multivocal Literature Review,” in *ACM Comput. Surv.*, Mar. 2024, doi: 10.1145/3652508.
- [19] A. V. Jha et al., “From theory to practice: Understanding DevOps culture and mindset,” in *Cogent Eng.*, vol. 10, no. 1, p. 2251758, 2023.
- [20] H. R. Kadaskar, “Unleashing the Power of Devops in Software Development,” in *Int. J. Sci. Res. Mod. Sci. Technol.*, vol. 3, no. 3, pp. 01–07, 2024.

- [21] Wikipedia, “DevOps toolchain,” 20 Jan 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=DevOps_toolchain&oldid=1197449470. [Accessed Apr 2025].
- [22] A. M. Davis, E. H. Bersoff and a. E. R. Comer, “A strategy for comparing alternative software development life cycle models,” in *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1453–1461, doi: 10.1109/32.6190, 1988.
- [23] A. Mishra and D. Dubey, “A comparative study of different software development life cycle models in different scenarios,” in *Int. J. Adv. Res. Comput. Sci. Manag. Stud.*, vol. 1, no. 5, 2013.
- [24] M. Kalske and others, “Transforming monolithic architecture towards microservice architecture,” Univ. Hels., 2017.
- [25] J.-P. Gouigoux and D. Tamzalit, “From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Gothenburg, France: IEEE, Apr. 2.
- [26] EMERALD Consortium, “D7.1 Project Manual and Quality Plan,” 2024.

APPENDIX A: Project Risks and impact in the DevOps Methodology

This section makes an analysis of some of the risks defined in the project – initially from the DoA [3] and then extended in D7.1 [26] - and the impact that the DevOps Methodology can have to mitigate them. It is important to note that these can evolve as part of the EMERALD *Task 7.2 Quality Assurance & Risk Management*.

Table 2. Risk and mitigation list

| Risk n. | Description | Proposed Mitigation Measures |
|---------|--|---|
| 1 | EUCS is not ready until 2026. | Not relevant. |
| 2 | Incompatibility between OSCAL and EMERALD (data import/export, modelling of security schemes). | Not relevant. |
| 3 | Users experience low usability. | WP4 will work in the UI/UX concept. The methodology contains activities to manage feedback from pilots. |
| 4 | EMERALD components are not able to be fully integrated. | The methodology should control which components are integrated and which components are not integrated. |
| 5 | Data set not sufficient for reaching TRL7 on the evidence collector components. | Not relevant. It should be controlled by the validation activities as part of other work packages. |
| 6 | The implementation does not cover all the use cases. | The methodology deployment activities should keep track of the use cases involved. |
| 7 | Underestimation of effort needed to complete activities. | The methodological approach promotes short cycles that will help to identify those situations faster. Besides, short cycles will focus on having running versions, and clearer view of what is missing. |
| 8 | Technology changes require significant redesign of the EMERALD architecture. | Integration tests will help to verify the redesigned elements. This will speed up the verification of refactored components as they are changed to the new architecture. |
| 9 | A partner fails to meet the obligations and becomes non-performing or even defaulting. | The methodological approach should promote multiple versions to have partial versions (instead of no versions) in that case. |
| 10 | Partner heterogeneity:
The different organizational and national cultures cause collaboration problems or conflicts in the project consortium | The methodology has been defined at the beginning of the project, and the clarity of the process paves the way for an easier collaboration. |
| 13 | Project execution risks:
a) key milestones are delayed
b) critical deliverables are delayed | The DevOps methodology, bringing together the work of developers, integrators and final users, helps to mitigate the possible delays. |
| 14 | Project key technologies,
development risks: | The DevOps Methodology can be easily adapted to cover other development languages or technologies. |

| | | |
|----|--|--|
| | <ul style="list-style-type: none">a) Key technologies or components are not available at the expected timeb) development takes longer than expectedc) wrong technology base is selectedd) lacking consensus on the technological approach between scientific partners | The DevOps Methodology automates the integration of the source code, and thus can speed up the deployment of delayed releases to make them available for the Use Cases. |
| 15 | Use case implementation is poor | The DevOps Methodology will produce three releases, as defined in the project plan, and the successive feedback can help making a better final implementation. Also, the IaC approach of the DevOps Methodology is a mitigation measure. |

APPENDIX B: Project Milestones from the DoA

This section includes the project milestone taken from the Description of Action (DoA) of EMERALD [3].

| # | Title | Validation | Month |
|---|--|---|-------|
| 1 | Project baselines and definition. Pilot set-up. Certification Graph Schema created. | <ul style="list-style-type: none"> • Project manual, public website. Defined the dissemination, communication and networking strategy. • Market analysis for EMERALD developed. • Data modelling for EMERALD components and initial design and requirements of the components. • CD/CI methodology for EMERALD defined. • Pilots' definition and evaluation strategy set up. | 9 |
| 2 | First release of the EMERALD components. | <ul style="list-style-type: none"> • EMERALD overall design specification and architecture • Initial prototypes of the main components of EMERALD. | 12 |
| 3 | First release of EMERALD integrated audit suite. First version of the EMERALD business models and plans, communication and dissemination report. | <ul style="list-style-type: none"> • Initial prototype of the EMERALD integrated solution with the functionalities implemented at M12. • First versions of the EMERALD business models, dissemination and communication reports. • Second version of EMERALD CD/CI methodology. | 18 |
| 4 | First implementation and evaluation of the first release of the EMERALD solution in the pilots. | <ul style="list-style-type: none"> • First implementation of the first release of the EMERALD tools in the use cases | 20 |
| 5 | Second release of EMERALD components. | <ul style="list-style-type: none"> • Second version of the EMERALD architecture. • Second releases of the main components of EMERALD. | 24 |
| 6 | Second release of EMERALD integrated audit suite. | <ul style="list-style-type: none"> • Second prototype of the EMERALD integrated solution with the functionalities implemented at M24. | 30 |
| 7 | Second implementation and evaluation of the second release of the EMERALD solution in the pilots. | <ul style="list-style-type: none"> • Second implementation of the second release of the EMERALD tools in the use cases. | 32 |
| 8 | Final release of EMERALD integrated audit suite. | <ul style="list-style-type: none"> • Final prototype of the EMERALD integrated solution with feedback from the second evaluation of the pilots. | 34 |
| 9 | Final evaluation report and impact analysis. Final version of the EMERALD business models and plans, communication, and dissemination report. | <ul style="list-style-type: none"> • Final evaluation and impact analysis from the pilots. • Final versions of the EMERALD business models dissemination and communication reports. | 36 |

APPENDIX C: Integration files

This annex provides additional technical details about the CI/CD strategies, such as the *renovate* mechanism, the semantic versioning configuration, the *Kustomize* approach and the *Docker Compose* files. We also include some CI/CD samples as reference, as they are placed in the private *GitLab* area.

C.1 – Renovate mechanism

The CaaS Framework evolves for two main reasons: the need to adjust the integration, or the need to update a specific component. The first case is expected to be the most common one in the initial phases of the project. The second case is expected to be more common in the last phases of the project, when the integration is more stable.

For this second case, we have implemented a *Renovate* mechanism that is based on the *GitLab* CI/CD pipeline. In the private area of the EMERALD *GitLab* repository, we have a project (*emerald/private/devops/renovate-agent*) that includes two main elements:

- A *GitLab* CI/CD pipeline that runs *Renovate* over a set of repositories
- A scheduled pipeline that runs the *Renovate* pipeline every week

Listing 4 shows the content of the *.gitlab-ci.yml* file of the *renovate-agent* project. The main elements of the file are the *include* element and the *RENOVATE_REPOSITORIES* variable. The *include* element includes the component *renovate-agent*, which is the main element of the pipeline. The *RENOVATE_REPOSITORIES* variable defines the repositories that will be updated by the *Renovate* mechanism:

- *emerald/private/devops/side-services*
- *emerald/private/devops/caas-framework*
- *emerald/private/devops/caas-framework-development*

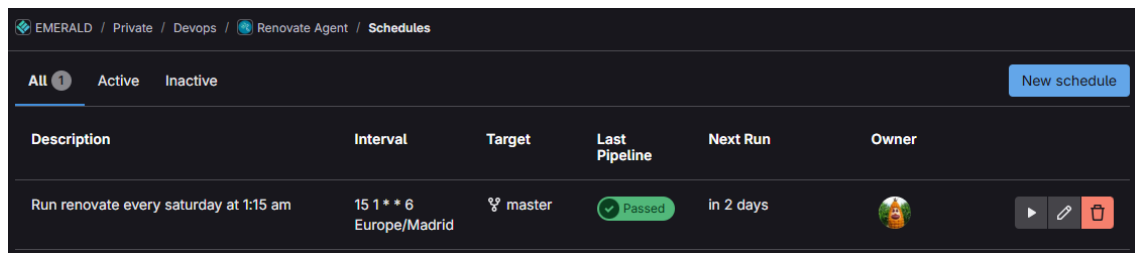
```
include:
  # renovate template
  - component: "git.code.tecnalia.dev/smartdatalab/public/ci-cd-components/renovate/gitlab-ci-renovate@master"
  inputs:
    renovate-validator-job-tags: ["docker"]
    renovate-depcheck-job-tags: ["docker"]

renovate-depcheck:
  rules:
    # not dry run on manual, schedule, pipeline or trigger
    - if: '$CI_PIPELINE_SOURCE == "schedule" || $CI_PIPELINE_SOURCE == "web" || $CI_PIPELINE_SOURCE == "pipeline" || $CI_PIPELINE_SOURCE == "trigger"'
      variables:
        RENOVATE_DRY_RUN: "false"
    - if: $RENOVATE_TOKEN

variables:
  RENOVATE_REPOSITORIES: '["emerald/private/devops/side-services", "emerald/private/Devops/caas-framework", "emerald/private/devops/caas-framework-development"]'
```

Listing 4. Renovate pipeline (*.gitlab-ci.yml*)

We configure the scheduled pipelines using the *GitLab* web interface (https://git.code.tecnalia.dev/emerald/private/devops/renovate-agent/-/pipeline_schedules). Figure 5 shows the current configuration of that schedule.




| Description | Interval | Target | Last Pipeline | Next Run | Owner |
|--|-----------------------------|----------|---------------|-----------|---|
| Run renovate every saturday at 1:15 am | 15 1 * * 6
Europe/Madrid | 🔗 master | ✅ Passed | in 2 days |  |

Figure 5. Renovate schedule

Another aspect of the *Renovate* configuration is the configuration of the `RENOVATE_REPOSITORIES`. This configuration is optional and is defined in a file called `renovate.json` located at the root of the repository. Listing 5 shows the configuration used in the side-services as example.

```
{
  "$schema": "https://docs.renovatebot.com/renovate-schema.json",
  "extends": ["config:base"],
  "packageRules": [
    {
      "matchManagers": ["gitlabci"],
      "automerge": true,
      "automergeType": "pr",
      "platformAutomerge": true,
      "rebaseWhen": "auto",
      "ignoreTests": true
    },
    {
      "matchManagers": ["kubernetes"],
      "semanticCommitType": "fix",
      "automerge": true,
      "automergeType": "pr",
      "platformAutomerge": true,
      "rebaseWhen": "auto",
      "ignoreTests": true
    }
  ],
  "kubernetes": {
    "fileMatch": ["architecture-and-data-modelling/.*\\.yaml$"]
  }
}
```

Listing 5. Content of the file `renovate.json`

This file configures the behaviour of the *Renovate* component. For example, in the side-services project we concentrate the *Renovate* in gitlabci and Kubernetes technologies. Besides, for the Kubernetes technology we configure it to check only files under the architecture and data modelling folder.

C.2 – Semantic Versioning Configuration

Another aspect is the semantic versioning configuration. Every time a new commit is pushed on components or integration projects, if the *Semantic Release* CI/CD component is included in the `.gitlab-ci.yaml` a new release is generated.

In a similar way to *Renovate*, *Semantic Release* has a default behaviour that can be configured with a `.releaserc.yaml` file. Listing 6 shows the `.releaserc.yaml` in the architecture and data modelling repository (emerald/private/architecture-and-data-modelling).

```
plugins:
  - '@semantic-release/commit-analyzer'
  - '@semantic-release/release-notes-generator'
  - '@semantic-release/gitlab'
  - '@semantic-release/changelog'
```

```
# emulates bumpversion (replaces version in pyproject.toml)
- - semantic-release-replace-plugin
  - replacements:
    - files:
      - index.html
    from:
      - 'release: *\d+\.\d+\.\d+'
    to: 'release: ${nextRelease.version}'
    countMatches: true
# git commit/push modified files (CHANGELOG.md & pyproject.toml)
- - '@semantic-release/git'
  - assets:
    - index.html
    - CHANGELOG.md
    # the commit MUST trigger a pipeline on tag (to perform publish jobs)
    # can be skipped on prod branch
    message: 'chore(semantic-release): release ${nextRelease.version} - [ci skip on prod]'
branches:
  - main
  - master
tagFormat: '${version}'
```

Listing 6. Content of the file `.releaserc.yaml`

In this configuration we use different plugins, the most interesting ones are:

- release notes generator: this includes notes in the release.
- changelog: this includes a changelog file in the repository.
- release replace plugin: this finds regular expressions and replaces them with text. It is useful to introduce the version in the configuration files of the code.

C.3 – Kustomize approach

In this section we provide additional details for the *Kustomize* structure used in the CaaS Framework. Figure 6 shows a visual description of the structure of the *Kustomize* configuration of the integrated version of framework.

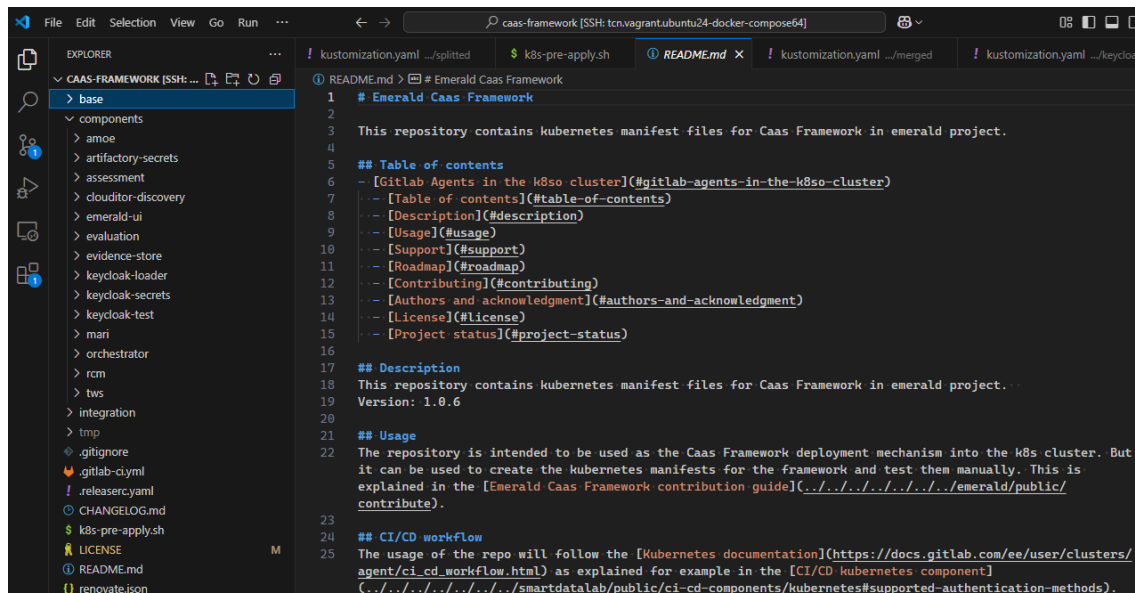


Figure 6. Kustomize main structure of the integrated CaaS framework

The main elements of the *Kustomize* component base structure are:

- Overlays (currently integration): describes the deployment of an environment, in this case the deployment environment.
- Components: includes the manifest for each component of the CaaS Framework. Most of the components are related to project functional components (AMOE, RCM, etc.).
- Base: describes the components that are included in the deployment. This base deployment is modified by the overlay to adapt to the different environments.

The files that describe the overlay are simple. Listing 7 shows an example. They basically point to the baseline to be used and over that base they describe the modifications to be applied. In the case below (Listing 7) it states that the namespace of the manifests merged in the base should be emerald-dev.

```
resources:
  - ../../base

namespace: emerald-dev
```

Listing 7. Kustomize integrate overlay

The files that describe the base are simple as well. Listing 8 shows an example. They basically enumerate the components to be included in the baseline. In the file below (Listing 8) it states the components and subcomponents that build the CaaS framework.

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

namespace: emerald-prod

components:
- ../../components/artifactory-secrets
- ../../components/keycloak-secrets
- ../../components/tws
- ../../components/rcm/keycloak/data
- ../../components/rcm/keycloak/loader
- ../../components/rcm
- ../../components/amoe/keycloak/data
- ../../components/amoe/keycloak/loader
- ../../components/amoe
- ../../components/orchestrator/keycloak/data
- ../../components/orchestrator/keycloak/loader
- ../../components/orchestrator
- ../../components/assessment/keycloak/data
- ../../components/assessment/keycloak/loader
- ../../components/assessment
- ../../components/emerald-ui/keycloak/data
- ../../components/emerald-ui/keycloak/loader
- ../../components/emerald-ui
- ../../components/clouditor-discovery/keycloak/data
- ../../components/clouditor-discovery/keycloak/loader
- ../../components/clouditor-discovery
- ../../components/evaluation/keycloak/data
- ../../components/evaluation/keycloak/loader
- ../../components/evaluation
- ../../components/evidence-store/keycloak/data
- ../../components/evidence-store/keycloak/loader
- ../../components/evidence-store
- ../../components/mari/keycloak/data
- ../../components/mari/keycloak/loader
- ../../components/mari
```

Listing 8. Kustomize base

The complexity comes in the components' specification. Inside each component folder, as shown in Figure 7, we will find multiple *Kubernetes* manifest (services, deployments, ingress, storage, jobs, configmaps, secrets, ...) that are joined in a *Kustomize* file.

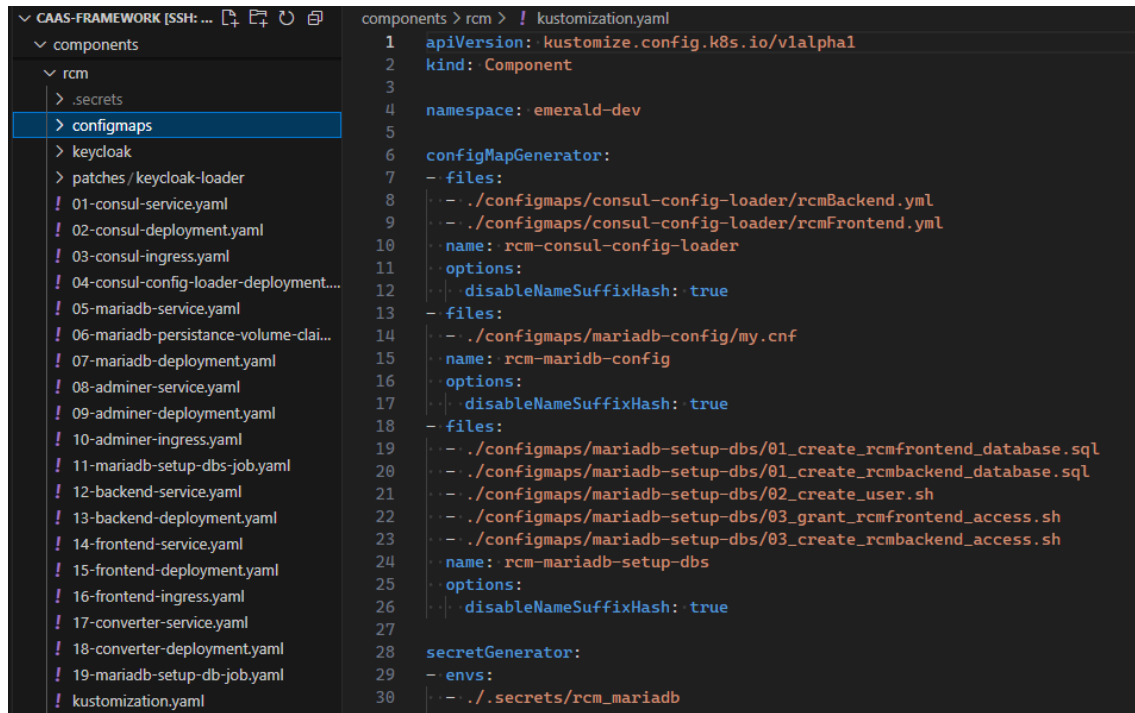


Figure 7. Kustomize component

The *Kustomize* file allows to create different elements for the component. Listing 9 shows an example of this file.

```
apiVersion: kustomize.config.k8s.io/v1alpha1
kind: Component

namespace: emerald-dev

configMapGenerator:
- files:
  - ./configmaps/consul-config-loader/rcmBackend.yml
  - ./configmaps/consul-config-loader/rcmFrontend.yml
  name: rcm-consul-config-loader
  options:
    disableNameSuffixHash: true
- files:
  - ./configmaps/mariadb-config/my.cnf
  name: rcm-mariadb-config
  options:
    disableNameSuffixHash: true
- files:
  - ./configmaps/mariadb-setup-dbs/01_create_rcmfrontend_database.sql
  - ./configmaps/mariadb-setup-dbs/01_create_rcmbackend_database.sql
  - ./configmaps/mariadb-setup-dbs/02_create_user.sh
  - ./configmaps/mariadb-setup-dbs/03_grant_rcmfrontend_access.sh
  - ./configmaps/mariadb-setup-dbs/03_create_rcmbackend_access.sh
  name: rcm-mariadb-setup-dbs
  options:
    disableNameSuffixHash: true

secretGenerator:
- envs:
  - ../secrets/rcm_mariadb
  name: rcm-mariadb
  options:
    disableNameSuffixHash: true
- envs:
  - ../secrets/rcm_backend
  name: rcm-backend
  options:
    disableNameSuffixHash: true
- envs:
  - ../secrets/rcm_frontend
```

```
name: rcm-frontend
options:
  disableNameSuffixHash: true
- envs:
  - ../secrets/rcm_converter
  name: rcm-converter
  options:
    disableNameSuffixHash: true

resources:
- ./01-consul-service.yaml
- ./02-consul-deployment.yaml
- ./03-consul-ingress.yaml
- ./04-consul-config-loader-deployment.yaml
- ./05-mariadb-service.yaml
- ./06-mariadb-persistence-volume-claim.yaml
- ./07-mariadb-deployment.yaml
- ./08-adminer-service.yaml
- ./09-adminer-deployment.yaml
- ./10-adminer-ingress.yaml
- ./11-mariadb-setup-dbs-job.yaml
- ./12-backend-service.yaml
- ./13-backend-deployment.yaml
- ./14-frontend-service.yaml
- ./15-frontend-deployment.yaml
- ./16-frontend-ingress.yaml
- ./17-converter-service.yaml
- ./18-converter-deployment.yaml
- ./19-mariadb-setup-db-job.yaml

components:
- patches/keycloak-loader
```

Listing 9. Kustomize RCM component

In this *Kustomize* code we can see some of the common *Kustomize* elements used:

- `configMapGenerator`: to load files as configmaps
- `secretGenerator`: to load secrets of different types
- `resources`: to add the manifests that declare the volume claims, services, deployments, ingresses, etc. to be created in *Kubernetes*
- `components`: to add other components

C.4 – Docker compose approach

In this section we present the structure of the local development framework used for the internal test based on *Docker Compose*. Figure 8 presents the structure of this framework.

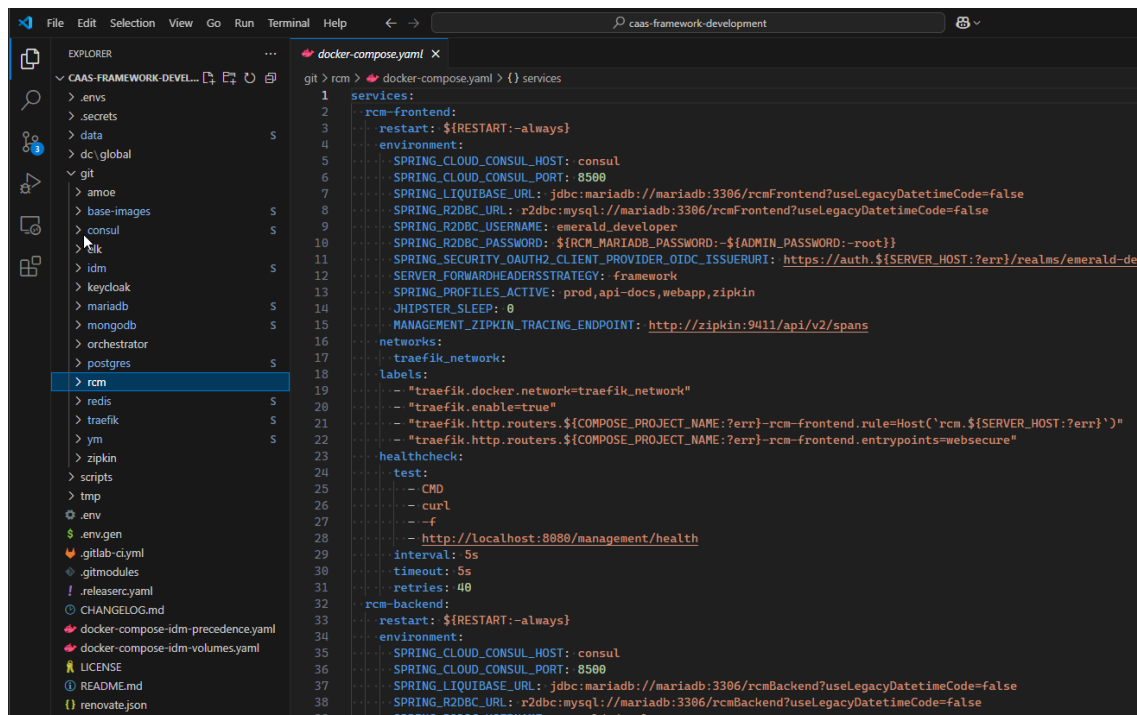


Figure 8. Docker Compose framework

The most remarkable elements in this structure are:

- components: In the git folder we include the projects of the different EMERALD components such as RCM. They are linked using the git submodule approach.
- data: In the data folder we point to the CaaS framework project with the git submodule mechanism.

The usage of this framework is simple once it is defined. If we are checking a component, we start it with:

```
docker compose up <component name>
```

To know the components included we issue a:

```
docker compose config --services
```

Figure 9 shows the current list of services supported in the local development environment.

```
/w/2023/g/em/pr/d/caas-framework-development git P master dc-config --services
/vagrant_project/scripts/tcn.docker-compose/tcn.docker-compose config --services
amoe-redis
postgres
keycloak-postgres-setup
keycloak
keycloak-setup
amoe-keycloak-setup
amoe-mongodb
amoe
amoe-mongodb-express
consul
rcm-consul-setup
ym
rcm-keycloak-setup
mariadb
rcm-mariadb-setup-dbs
rcm-frontend
rcm-backend
rcm-mariadb-setup-db
rcm-converter
amoe-redis-commander
zipkin
```

Figure 9. Local environment services

For example, if we are analysing the rcm-backend we just need to issue:

```
docker compose up rcm-backend
```

This will start the dependent services and once all the prerequisites have been started the rcm-backend will start, as shown in Figure 10.

The advantage appears when we are testing the effect of a small change. We can perform a change in the inner services, such as changing a property in the rcm-backend code (e.g., a description, as shown in Figure 10).

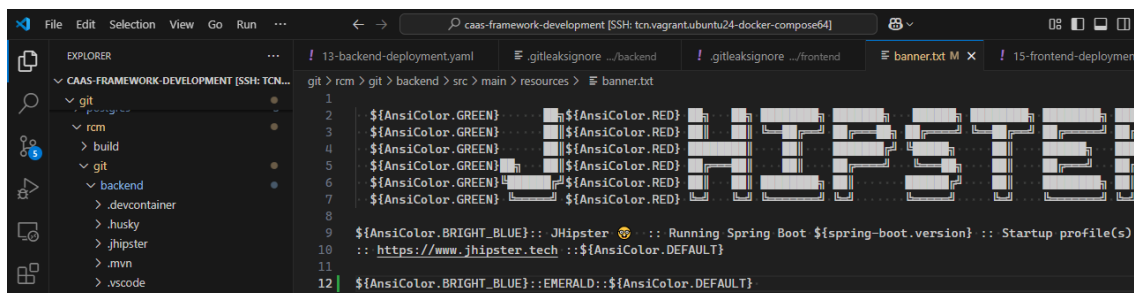


Figure 10. RCM change

Once the change has been made, to test again we only have to issue:

```
docker compose build rcm-backend
docker compose up rcm-backend
docker compose log rcm-backend
```

C.5 – CI/CD Examples

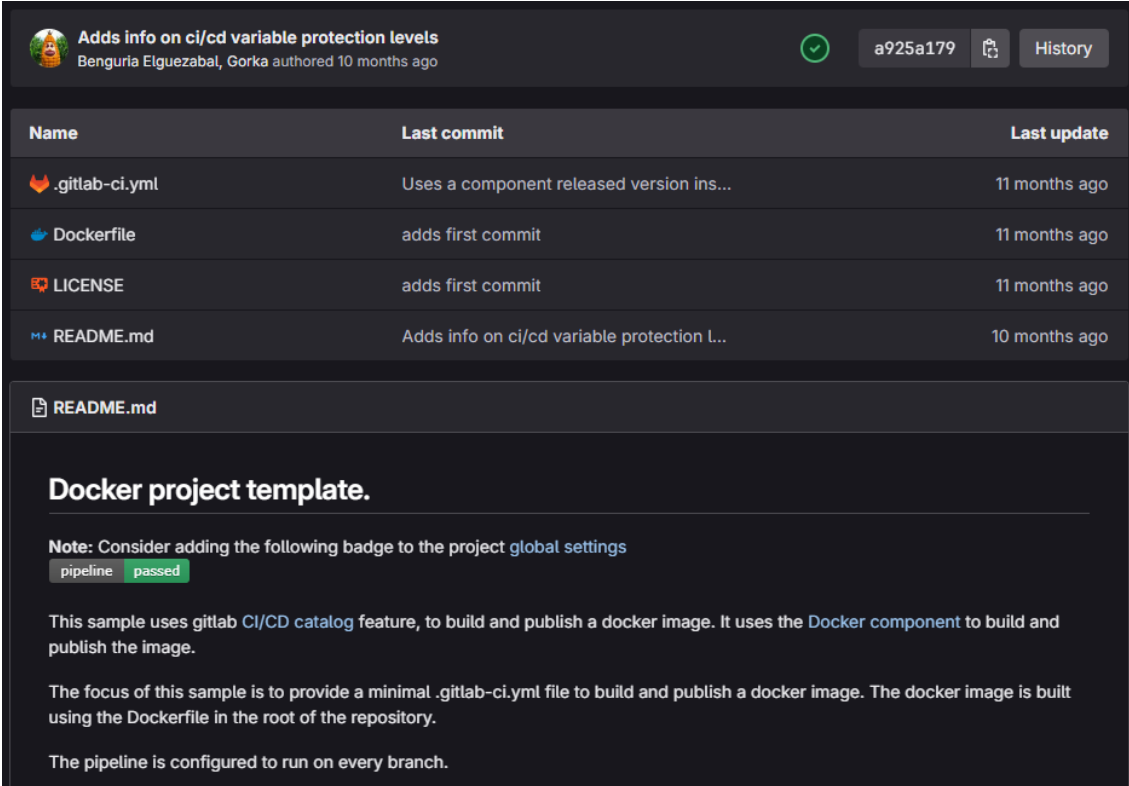
In the internal area of the GitLab repository there are several examples on how to apply the CI/CD Components. Here we describe some of them:

- *Docker*
- *Semantic Versioning*

Figure 11 shows the details of a *Docker* example that includes four files:

- `.gitlab-ci.yml`: This is the most important file that will be explained below.

- Dockerfile: This is the main *Docker* artifact. We added a very simple *Docker* file that adds some packages to a base Ubuntu image.
- LICENSE: This is a must have for any kind of repository.
- README.md: This explains the repository and its purpose.



| Name | Last commit | Last update |
|----------------|---|---------------|
| .gitlab-ci.yml | Uses a component released version ins... | 11 months ago |
| Dockerfile | adds first commit | 11 months ago |
| LICENSE | adds first commit | 11 months ago |
| README.md | Adds info on ci/cd variable protection L... | 10 months ago |

README.md

Docker project template.

Note: Consider adding the following badge to the project global settings

pipeline passed

This sample uses gitlab CI/CD catalog feature, to build and publish a docker image. It uses the Docker component to build and publish the image.

The focus of this sample is to provide a minimal .gitlab-ci.yml file to build and publish a docker image. The docker image is built using the Dockerfile in the root of the repository.

The pipeline is configured to run on every branch.

Figure 11. Docker CI/CD example

The focus of the example is to provide information on the *GitLab* CI/CD configuration. Listing 10 shows the content of the file.

```
include:
  - component: git.code.tecnalia.dev/smartdatalab/public/ci-cd-
    components/docker/gitlab-ci-docker@5
  inputs:
    snapshot-image: $CI_REGISTRY_IMAGE/snapshot:$CI_COMMIT_REF_SLUG
    release-image: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_NAME
    kaniko-build-job-tags: ["docker"]
    docker-publish-job-tags: ["docker"]
    # https://github.com/jfrog/artifactory-user-
    plugins/blob/master/cleanup/cleanDockerImages/README.md
  metadata: >-
    --label org.opencontainers.image.url=${CI_PROJECT_URL}
    --label org.opencontainers.image.source=${CI_PROJECT_URL}
    --label org.opencontainers.image.title=${CI_PROJECT_PATH}
    --label org.opencontainers.image.ref.name=${CI_COMMIT_REF_NAME}
    --label org.opencontainers.image.revision=${CI_COMMIT_SHA}
    --label org.opencontainers.image.created=${CI_JOB_STARTED_AT}
    --label com.jfrog.artifactory.retention.maxCount=4
  prod-publish-strategy: auto
  hadolint-disabled: true
  healthcheck-disabled: true
  trivy-disabled: true
  sbom-disabled: true
variables:
  CI_REGISTRY_IMAGE: emerald-docker-dev-local.artifact.tecnalia.dev/template-docker
  # CI_REGISTRY_USER defined in GitLab CI/CD settings
  # CI_REGISTRY_PASSWORD defined in GitLab CI/CD settings
```

Listing 10. CI/CD for docker generation

In the `.gitlab-ci.yml` code above you can see the usage of the release 5 *Docker* component (<https://git.code.tecnalia.dev/smartdatalab/public/ci-cd-components/docker/gitlab-ci-docker@5>). This means that the last release 5 will be applied. The component behaviour is controlled with input parameters, where we can:

- Specify the names of the images
- Filter the runners applicable for each of the jobs
- Add metadata to the image before publishing into *Artifactory*
- Select the building engine
- Enable or disable additional test and features.

The example is used internally, as well as the example pipelines (see Figure 12).

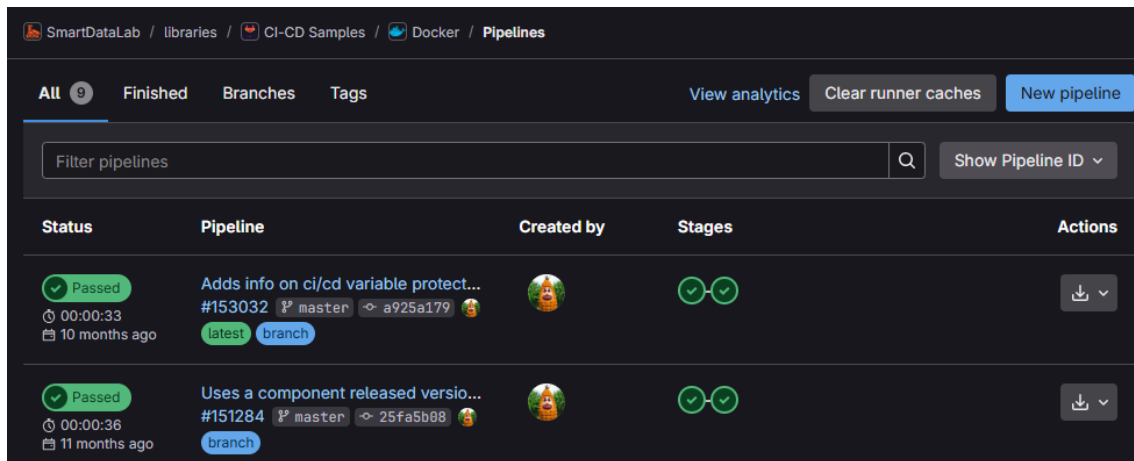


Figure 12. docker CI/CD example pipelines

We can also see the details of each stage (see Figure 13).

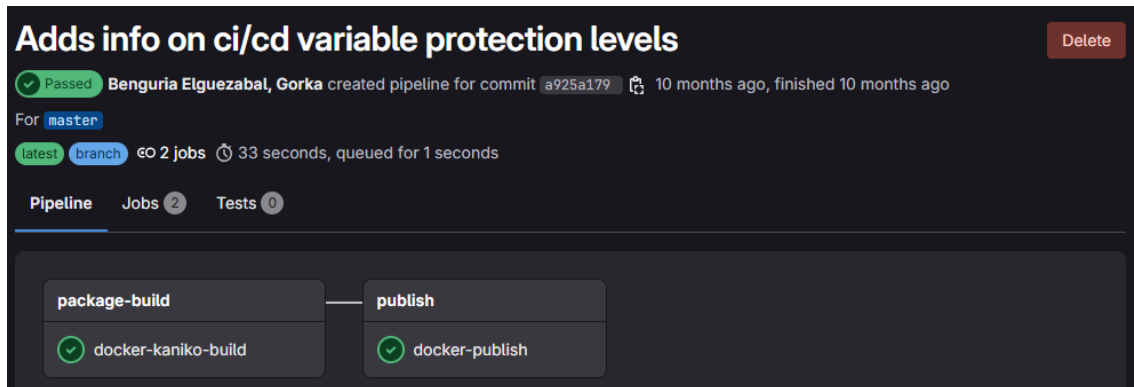



Figure 13. Docker CI/CD stages detail

Semantic Release examples follow a similar structure (see Figure 14). This example builds on the *Docker* example by adding semantic versioning capability.




Adds info on ci/cd variable protection levels
 Benguria Elguezal, Gorka authored 10 months ago
 95704211
History

| Name | Last commit | Last update |
|----------------|--|---------------|
| .gitlab-ci.yml | Merge branch 'master' of git.code.te... | 11 months ago |
| Dockerfile | adds first commit | 11 months ago |
| LICENSE | adds first commit | 11 months ago |
| README.md | Adds info on ci/cd variable protectio... | 10 months ago |

README.md

Semantic Release project template.

Note: Consider adding the following badge to the project [global settings](#)

This sample uses [gitlab CI/CD catalog](#) feature, to release using a [semantic versioning](#) approach. It uses the [Semantic Release component](#) to control the releases applying the semantic versioning approach [through the analysis of the commits](#).

semantic-release automates the whole package release workflow including: determining the next version number, generating the release notes, and publishing the package.

This removes the immediate connection between human emotions and version numbers, strictly following the [Semantic Versioning specification](#) and communicating the impact of changes to consumers (<https://github.com/semantic-release/semantic-release>).

The project uses the [Docker sample](#) as a base, and adds the semantic release component to the pipeline.

Figure 14. Semantic release CI/CD example

The focus of this example is to provide information on the GitLab CI/CD configuration. Listing 11 shows the content of the file.

```
include:
  - component: git.code.tecnalia.dev/smartdatalab/public/ci-cd-
    components/docker/gitlab-ci-docker@5
  inputs:
    snapshot-image: $CI_REGISTRY_IMAGE/snapshot:$CI_COMMIT_REF_SLUG
    release-image: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_NAME
    kaniko-build-job-tags: ["docker"]
    docker-publish-job-tags: ["docker"]
    # https://github.com/jfrog/artifactory-user-
    plugins/blob/master/cleanup/cleanDockerImages/README.md
  metadata: >-
    --label org.opencontainers.image.url=${CI_PROJECT_URL}
    --label org.opencontainers.image.source=${CI_PROJECT_URL}
    --label org.opencontainers.image.title=${CI_PROJECT_PATH}
    --label org.opencontainers.image.ref.name=${CI_COMMIT_REF_NAME}
    --label org.opencontainers.image.revision=${CI_COMMIT_SHA}
    --label org.opencontainers.image.created=${CI_JOB_STARTED_AT}
    --label com.jfrog.artifactory.retention.maxCount=4
  prod-publish-strategy: auto
  hadolint-disabled: true
  healthcheck-disabled: true
  trivy-disabled: true
  sbom-disabled: true
  - component: git.code.tecnalia.dev/smartdatalab/public/ci-cd-components/semantic-
    release/gitlab-ci-semrel@3
  inputs:
    release-disabled: false
    semantic-release-job-tags: ["docker"]
    auto-release-enabled: true
```

```
branches-ref: "/^(master|main)$/"

variables:
  CI_REGISTRY_IMAGE: emerald-docker-dev-local.artifact.tecnalia.dev/template-docker
  # CI_REGISTRY_USER defined in GitLab CI/CD settings
  # CI_REGISTRY_PASSWORD defined in GitLab CI/CD settings
```

Listing 11. CI/CD for semantic release generation

In the `.gitlab-ci.yml` code above you can see how we extend the previous *Docker* example with five lines to declare the component's *Semantic release* and parametrize its behaviour.

We have also tested it, and you can see the pipelines and details of their stages. In the image below (Figure 15) we can see that it is similar to the *Docker* pipeline but with an additional job.

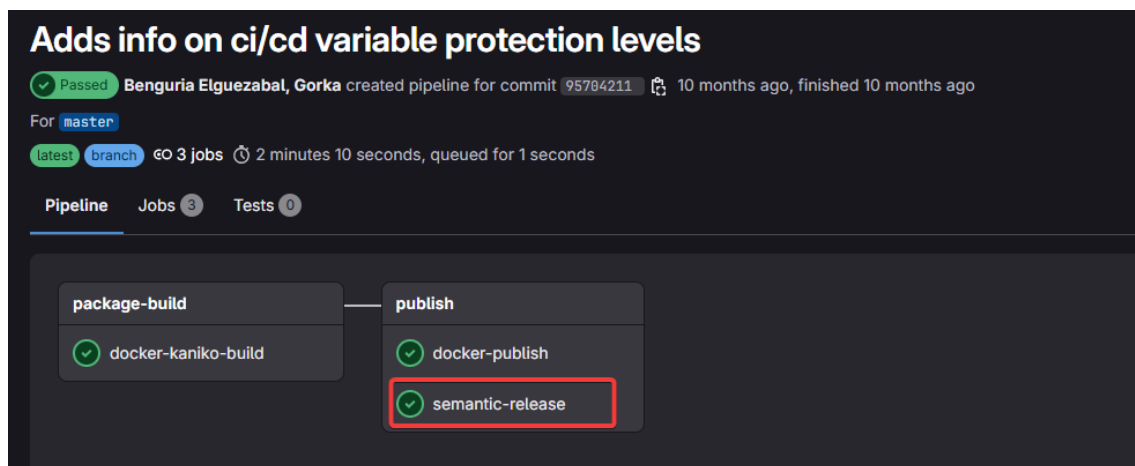


Figure 15. Semantic release CI/CD stages detail