This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nº 101017331

D3.1 Operational framework

Deliverable No.	D3.1	Due Date	30/06/2021
Description	Establishes the infrastructure, protocols, structures and operations for development and operation of the ODIN platform, to be used in WP3-5.		
Type	Report	Dissemination Level	PU
Work Package No.	WP3	Work Package Title	Platform integration, Privacy, Security and Trust + knowledge + cognition
Version	1.0	Status	Final

Authors

Name and surname	Partner name	e-mail
Ilias Kalamaras	CERTH	kalamar@iti.gr
Dimitrios Giakoumis	CERTH	dgiakoum@iti.gr
Konstantinos Votis	CERTH	kvotis@iti.gr
Giuseppe Fico	UPM	gfico@lst.tfo.upm.es
Alejandro Medrano	UPM	amedrano@lst.tfo.upm.es
Eugenio Gaeta	UPM	eugenio.gaeta@lst.tfo.upm.es
Álvaro Belmar	UPM	abelmar@lst.tfo.upm.es
Ezequiel Simeoni	UPM	esimeoni@lst.tfo.upm.es
Pilar Sala	MYS	psala@mysphera.com
Jesús Gago Centeno	INETUM	jesus.gago@inetum.world
Marta Millet	ROB	mmillet@robotnik.es
Ernesto Iadanza	UoW	ernesto.iadanza@warwick.ac.uk

History

Date	Version	Change
25/05/2021	0.1	Initial draft containing table of contents and first skeleton content.
02/06/2021	0.2	First draft content.
14/06/2021	0.3	Added content in all sections.
16/06/2021	0.4	Added content in all sections.
16/06/2021	0.5	Integrated input from INETUM.
18/06/2021	0.6	Integrated input from UoW and UPM, addressed partners' comments and added further content.
21/06/2021	0.7	Added further content in all sections.
21/06/2021	0.8	Version ready for peer-review.
28/06/2021	0.9	Additional input for peer-review.

28/06/2021	0.10	Integrated input from peer-review and addressed comments from other partners.
29/06/2021	0.11	Minor corrections.
29/06/2021	0.12	Version ready for quality check.
30/06/2021	0.13	Addressed comments of quality check.
30/06/2021	1.0	Final version.

Key data

Keywords	DevOps, operational framework, continuous development/integration
Lead Editor	Ilias Kalamaras (CERTH)
Internal Reviewer(s)	Marta Millet Pascual-Leone (ROB) Pablo Lombillo Biosca (MYS)

Abstract

D3.1 Operational framework describes the software infrastructure used to support continuous development and integration of software components developed in ODIN. The infrastructure involves source code versioning, build tools, component repositories, and automatic deployment tools for installation and use at the pilot sites. The goal of the infrastructure is to create a continuous workflow from developer to end user, so that any new functionalities or bug fixes are available to the end user as soon as possible and with minimum effort.

Statement of originality

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

Table of contents

List of tables

List of figures

1 Introduction

This deliverable describes ODIN's operational framework for continuous development, integration and delivery of software components, collectively called DevOps. The document covers a continuous workflow from developer to end-user, including source code management and version control, build automation tools, containerization tools, software testing frameworks, component release guidelines, component deployment, deployment management, as well as the tools that will be used to automate the execution of the steps in this workflow.

The deliverable provides a description of the infrastructure that will be implemented in ODIN regarding DevOps, including version control servers, Docker registries, deployment management servers, orchestration services and security provisioning. It provides a description of popular alternatives for the tools in each step, from which the ones to be used in ODIN are selected. It also provides information about best practices for the different steps to facilitate developers and ensure high-quality software production.

The deliverable is meant to provide guidelines to the technical members of the ODIN consortium regarding the services provided by ODIN's DevOps infrastructure, and about how to use them during development and deployment. Sections 2 to 9 cover the main steps in the DevOps workflow. Each section is structured in the same high-level manner:

- An introduction to the type of activities involved in the step to be described
- A presentation of the available tools to facilitate these activities
- A section named "ODIN guidelines", describing the tools and procedures that will be used and followed within ODIN regarding the corresponding activities, as well as the infrastructure that will be setup to manage them.

A reader wishing to find guidance about how each step should be addressed within ODIN may start by consulting these "ODIN guidelines" sections, which can be used as a guide with references to the other parts of the deliverable for details, where necessary. A summary of the whole ODIN DevOps infrastructure, with links to the appropriate guidelines is provided in 10.

This deliverable will be shared with the consortium and will establish the DevOps guidelines that will be maintained during the project's lifetime. There will be no other version of this deliverable. However, the guidelines contained in this version will be transferred to the project Wiki (once it is released, as part of the activities of T3.4). The version uploaded to the Wiki will act as a running document shared among the consortium that will be updated whenever changes occur. Such changes may include changes in DevOps domain names, or switching to a different tool for a particular step if problems arise in practice. In such cases, any modifications will be reported in the Wiki and disseminated to the technical members of the consortium through appropriate communication channels, such as project meetings and mailing lists.

1.1 Deliverable context

Table 1 provides an overview of the context of the current deliverable, in relation to the project objectives and foreseen results.

Table 1: Deliverable context.

PROJECT ITEM	RELATIONSHIP		
Objectives	The deliverable is relevant to ODIN's Objective 1, as it describes and defines the basis for the development and deployment of the components that comprise the foreseen open and secure decentralized ODIN platform.		
Exploitable results	There is no specific contribution to any exploitable results. Instead, the infrastructure presented hereby will be used as the basis for the development of potentially exploitable components.		
Workplan	D3.1 is attributed to the tasks of WP3, Platform integration, Privacy, Security and Trust + knowledge + cognition. Specifically, the task involved in the preparation of this deliverable is T3.1, DevOps and infrastructure.		
Milestones	D3.1 is a key deliverable of the PREPARATION (MS1) and IMPLEMENTATION (MS3) phases of the project.		
Deliverables	D3.4 – D3.6	Privacy Security and Trust report	Regarding security mechanisms
	D3.7 – D3.9	Technical Support Plan and Operations	Regarding component documentation and feedback collection.
	D3.10 – D3.12	ODIN platform	Regarding the application of DevOps in the development of the ODIN platform.
	D7.2 – D7.7	KPI Evolution Report (I to IX)	Regarding the collection of KPIs about DevOps activities.
	D7.9	Pilot Studies Evaluation Results and sustainability	Regarding component evaluation results of unit/integration testing.
Risks	The guidelines provided in this deliverable can help in minimizing the following risks identified in the Grant Agreement: • Technologies not available in time • Technical problems during component/module development • Complexity of unification procedure The described DevOps guidelines provide a continuous development/integration infrastructure and best practices that can assist in delivering components in time, reducing technical problems in component development and deployment, and reducing complexity of deployment follow-up through a continuous delivery pipeline.		

1.2 DevOps overview

DevOps is a term to describe a culture where the fields of software development and operation are brought closer together, facilitating and accelerating the release of new functionalities[1].

DevOps involves two key ideas:

- Adopting practices for developing high-quality, production-ready and easy to maintain software, such as naming conventions for easy collaboration, unit testing for quickly spotting bugs, comprehensive documentation, etc.
- Making use of automation tools for automatic building, distributing and deploying software, so that new functionalities and changes are propagated as soon as possible to the end users for operation.

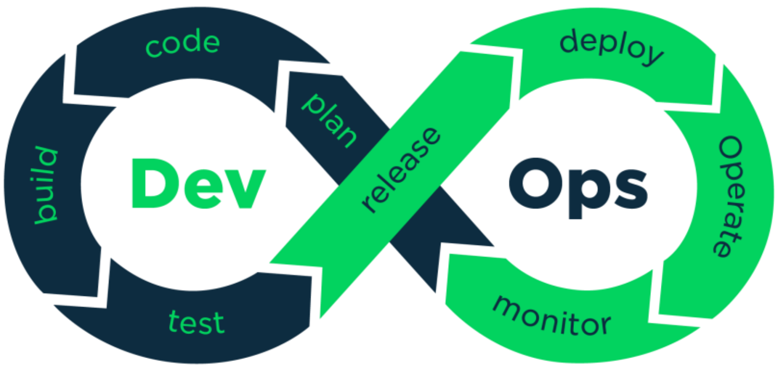


Figure 1: The general DevOps workflow. Source devopedia.org.

The steps involved in a continuous integration/continuous delivery (CI/CD) workflow are mainly the following, as shown in Figure 2:

- Source code management: Structuring and documenting source code, managing versions, collaborating, etc.
- Building automation: Building source code into executable applications.
- Testing: Writing and executing unit and integration tests to check the proper functioning of the application.
- Release: Making the application available to end-users.

[1] Devopedia, https://devopedia.org/devops. Last access June 2021.

- Deployment: Installing the application to the target execution environment, e.g., at a pilot site.
- Execution / monitoring: Running the application at the target environment and collecting feedback about its use and malfunctions
- Orchestration: Automating the whole process from developer to target execution environment.

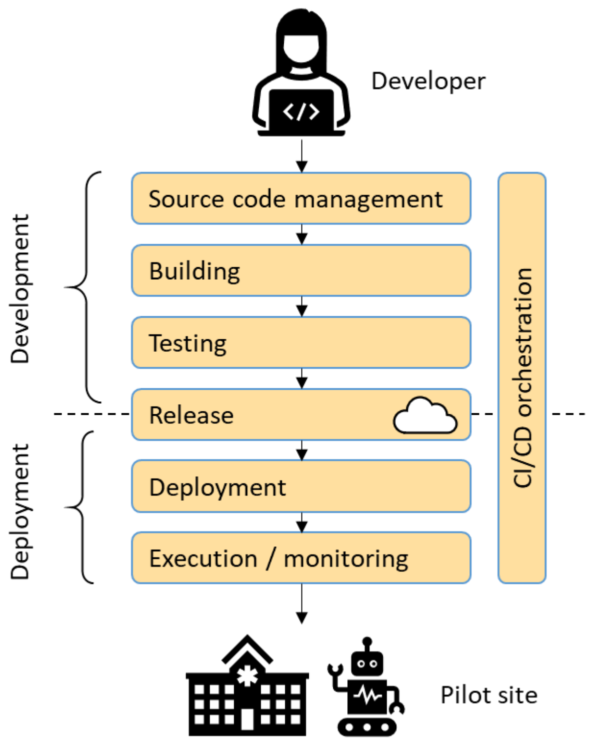


Figure 2: High-level overview of the CI/CD workflow to be used in ODIN.

These steps create a continuous workflow from the developer to the target execution environment. The orchestration mechanisms are responsible for automating this procedure, so that changes in the source code made by the developer are automatically propagated through all steps of the workflow and end up in a new software version running at the pilot site.

In a more detailed view, in ODIN, each of these steps will be handled by a particular DevOps component, described in the following sections of this deliverable. An overview of the specific components to handle the different parts can be seen in Figure 3. On one end, the developer is responsible for providing the source code, along with configuration files needed for properly building, containerizing and deploying the built application. The source code and the configuration files are managed by a GitLab source code management server, split into two parts: one handling the source code and building instructions and another handling component deployment. As a next step, the source code is built into executable applications and containerized into Docker images, ready to be installed in any execution environment (see Section 3.1.7 for details). The created Docker images are released to ODIN's Docker registry, at which point the development phase is complete.

During the deployment phase, the instructions provided by the developer are used to compose the available Docker images into complete applications that are started in the environment of the pilot site. The composition process and the monitoring of the running services are managed through an appropriate dashboard.

The orchestration and automation of the whole workflow from developer to pilot site is managed by the DevOps administrator, with the use of Jenkins orchestration software. The DevOps administrator provides a set of Jenkins instruction files, specifying the series of steps needed to run each time a component's source code is altered by the developer. The Jenkins instruction files are managed in a separate part of ODIN's GitLab. The instructions are executed by a Jenkins server, automating the whole process.

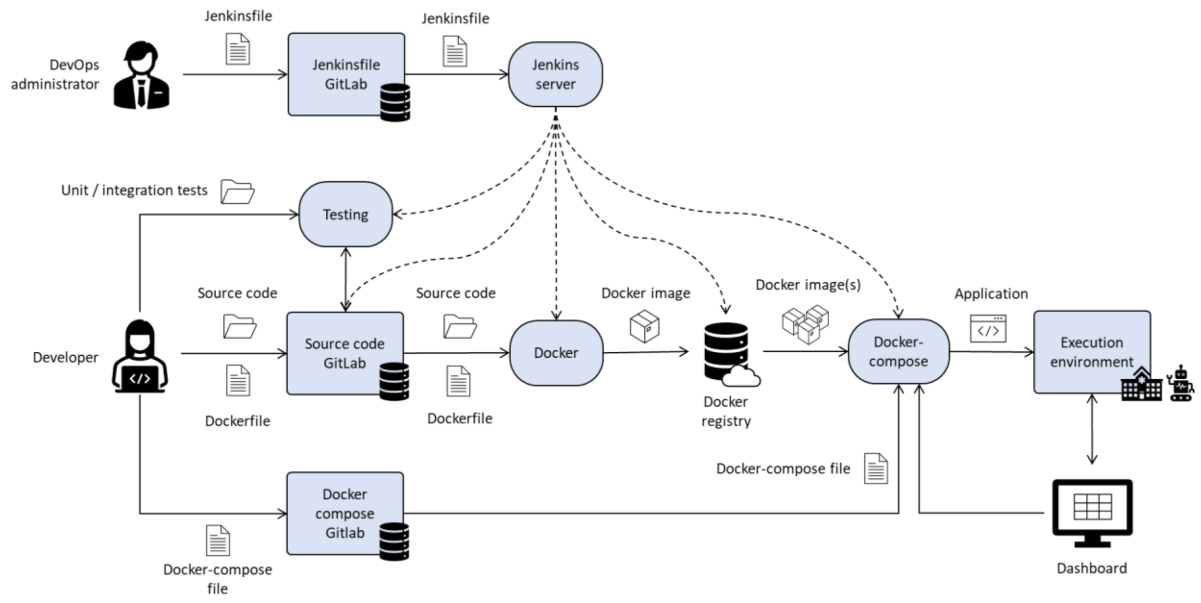


Figure 3: Detailed overview of the ODIN DevOps workflow.

In the following sections, each step of this procedure is described in detail. Each section covers an overview of the activities taking place at each step, the available tools to accomplish these activities, and the way each step will be handled in ODIN, providing thus a set of guidelines for developers and maintainers to follow.

2 Source code management

The starting point for the development lifecycle of a software component is the source code written by the developer. This section describes the conventions to be used in ODIN for source code organization, as well as the tools that will be used for source code storage, management, versioning and collaborative authoring.

2.1 Source code versioning

Source code versioning tools facilitate keeping track of changes in the source code during the course of development. They relieve the developer from the need to keep backups of different versions of a project as it is being developed, as new functionality is being added and as bugs are fixed. Whenever a particular change has been made to address an issue, the code can be committed to the versioning tool, which keeps track of all previous commits. If the new change is found to be problematic, versioning tools allow the developer to roll back to previous versions, bringing the whole project to an earlier state, so that they can bring the application to a working state and discover what went wrong. Versioning can be applied not only to source code, but any type of file, e.g., documents, configuration files, etc., whenever one needs to keep track of the project versions. Source code versioning tools are the standard way to keep track of source code versions in production software.

The overall manner in which versioning tools work is the following. The developer starts *tracking* a particular directory, e.g., the root directory of a new project. The developer can make changes in the directory, e.g., add/modify/delete files. At any point during the project development, the developer can choose to *commit* the changes to the versioning tool, which stores the current state of the directory and can tell what changed since the previous version. The complete history of commits is maintained by the versioning tool. Whenever the developer wishes to roll back to a previous version, they can *checkout* a previous commit, which brings the whole directory to the state it was at that commit, so that the developer can examine its contents as they were at that point.

Apart from tracking the history of changes, versioning tools also provide two other major functionalities:

- *Branching*, i.e., creating different paths of development starting from the same commit, which can later be merged if needed. This is e.g. used to maintain a master branch of the stable version of a piece of software, while experimental branches are initiated from different points in the master path, in order to develop and test new functionality. Once the new functionality is finished, the experimental branch can be merged with the master branch to create a new stable version.

- *Pushing to a remote repository*, i.e., uploading the whole tracked history to an online repository. On the one hand, this is used to create a backup of the whole project history in a remote location. On the other hand, this is used as a means for collaboration among several developers. Different developers can work in different parts of a project, pushing their changes in the common project repository.

2.2 Source code versioning tools

There are several tools that provide source code versioning functionalities. Versioning tools follow two general paradigms: centralized and distributed version control. In centralized systems, the source code and its history are stored in a central server, with each developer communicating with the server in order to get the latest version and commit changes. In distributed systems, each developer maintains a complete copy of all source code and its history. Changes are committed locally, and they can be pushed to remote repositories, for storage and sharing with others. Both paradigms have their advantages and disadvantages. Centralized systems make it easier to manage a project, but are time consuming since it requires a constant connection with the server. Distributed systems are fast but may make it difficult to coordinate work of many developers. Early version control systems, such as Subversion, followed the centralized paradigm. However, the advantages of distributed systems, such as Git, have prevailed over the years, and they are those that are mostly in use today. Here we briefly review some characteristic and widely used version control systems.

2.2.1 Subversion (SVN)

Apache Subversion[2], or SVN, is a widely used centralized version control system. Centralized means that the whole repository is stored in a central SVN sever, and each developer can contribute to the repository by committing changes to specific parts. One of the main advantages of a centralized version control system is the ease of managing the repository, as there is a single point where all code is gathered, and the administrator can have the full overview. Another advantage is that collaborating developers can each have an overview of all the other developers' work.

Disadvantages include the need to be connected to the server in order to commit changes, which makes it difficult to work in case of server failure. Committing changes to the server may also induce latencies in the development workflow, as uploading large files may be time-consuming. These problems are reduced in distributed version control systems, since commits are made locally and may only be pushed to the repository after several changes have been made. Another drawback of SVN is the fact that creating new branches is an expensive procedure, requiring several file copies, discouraging developers from following branching-based workflows.

Subversion offers a command line interface with commands for committing and pushing code and managing repositories. GUI tools can also be used as front-ends to the SVN system, such as TortoiseSVN[3] and RapidSVN[4], as well as add-ons and extensions to popular IDEs and code editors.

[2] Apache Subversion, https://subversion.apache.org/ Last access June 2021.

[3] TortoiseSVN, https://tortoisesvn.net/ Last access June 2021.

[4] RapidSVN, https://rapidsvn.org/ Last access June 2021.

2.2.2 Mercurial

Mercurial[5] is a distributed source code management system that aims at fast management of large projects. Distributed means that local copies of the complete repositories are stored in each developer's machine, so that all changes are first made locally, and pushed to the remote repositories when needed.

The fast tracking of source code with local commits and the ability to commit changes even without an Intenet connection are significant advantages of Mercurial over SVN, which is a centralized system. Mercurial also allows to extend its functionalities through extensions. These extensions may e.g. provide access control mechanisms, usage statistics, notifying developers through e-mails, etc. However, the branching system of Mercurial is still not quite easy to use making it a bit hard for developers to securely manage branches.

Mercurial offers a simple set of commands to use, and is thus easy for new users to learn. Graphical user interfaces are also available from Mercurial, such as TortoiseHG[6], and it has also been integrated in popular IDEs and editors, such as Eclipse, NetBeans, Visual Studio, Emacs and Vim.

2.2.3 Git

Git[7] is an open-source versioning system and is one of the most popular versioning systems. It is a distributed version control system, similar to Mercurial, meaning that each developer working on a project maintains a copy of the whole project repository, thus reducing the risk of failure. At each commit, Git stores a snapshot of the directory structure at the time of the commit, i.e., copies the current versions of all files, apart from files which have not changed, for which only a link to the previous version is stored. Git achieves high speeds in versioning, since all commits are made locally, and only pushes the changes to a remote repository upon request by the developer. Git also focuses on easy branching, since a new branch does not create copies of any files, just creates pointers to existing snapshots. This allows developers to create and merge branches often, encouraging experimentation.

Git offers a rich command line interface, with commands to track files, commit changes, push to remote repositories, manage repositories, checkout previous versions, create and manage branches, etc. Graphical User Interfaces (GUIs) are also available, such as TortoiseGit[8] and Git Extensions[9], which allow a visual management of repositories. There are also several extensions

[5] Mercurial, https://www.mercurial-scm.org/ Last access June 2021.

[6] TortoiseHG, https://tortoisehg.bitbucket.io/ Last access June 2021.

[7] Git, https://git-scm.com/ Last access June 2021.

[8] TortoiseGit, https://tortoisegit.org/ Last access June 2021.

[9] Git Extensions, https://gitextensions.github.io/ Last access June 2021.

to popular editors and Integrated Development Environments (IDEs), such as Visual Studio Code, Eclipse, Vim, etc. for managing repositories directly from within the IDE.

The advantageous characteristics of Git and its popularity have led to the design of development workflows based on its versatile branching scheme, as well as to the creation of widely used online Git repositories. These are briefly summarized below.

2.2.3.1 GitFlow

GitFlow[10] is a development flow, conceived by Vincent Driessen, which describes a very precise branching model built around the concept of software release. This flow is designed to make the most out of the potential of the Git versioning software, but conceptual affinities can also be useful for managing the work with other software dedicated to the same functionality.

The flow described in GitFlow is aimed at maintaining a clean implementation history, where a release communicates to all users the presence of a new version of the product, defined by a specific changelog consisting of new features and fixes. An example of a GitFlow workflow is shown in Figure 4.

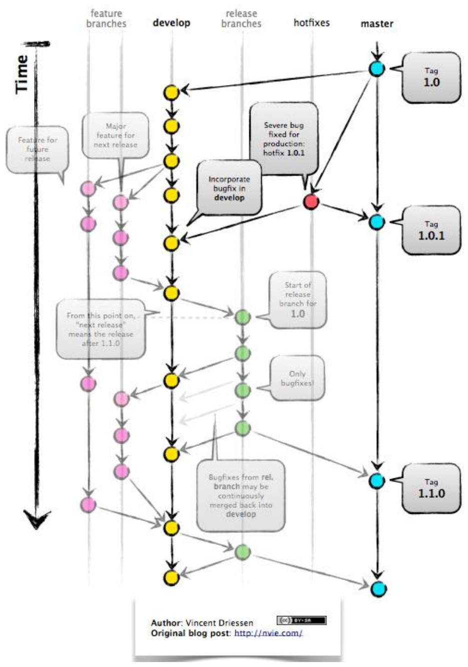

feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 1.0.1
Incorporate bugfix in develop
Tag 1.0.1
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/

Figure 4: Example of a GitFlow workflow.

GitFlow allows to:

[10] GitFlow, https://datasift.github.io/gitflow/IntroducingGitFlow.html Last access June 2021.

- Develop in parallel: new developments are organized in feature branches and are merged into the main code only when the team deems it ready for release. This allows the developers to change tasks without problems.
- Increase collaboration: feature branches allow multiple developers to work on a single feature, as it is like a sandbox and all developments are carried out so that it is brought into production. This also allows one to verify the work done by individual developers on a feature.
- Have a release staging area: new developments are merged into a develop branch, which effectively represents a staging area for all developments that have not yet been released. This means that when a release occurs, the latter has all the developments in the develop branch inside.

Support for emergency fixes: there is support for hotfix branches which are nothing more than branches of a release. In this way, the merge will take place directly in the release branch, allowing fast fixes that should also be merged in the development branch.

2.2.3.2 Online Git repositories

Popular online repositories for Git projects include GitHub[11] and Bitbucket[12]. They can both be used to setup remote repositories for projects and link them to local Git repositories, so that developers can push to these online repositories. Online Git repositories allow the dissemination of source code to the community, so that other developers can use and modify it. They also facilitate collaboration among teams of developers. Both GitHub and Bitbucket provide public or private repositories, and they provide different storage capacity and functionalities according to their pricing plans.

An alternative to using public repositories such as the above is for developer teams to setup their own internal Git repository at a dedicated server. GitLab[13] is a popular choice for such a scenario. It provides the infrastructure for hosting a Git repository where teams can push their code. GitLab can be configured for the needs of a particular team, supporting any number of users and controlling the available storage space. It also provides DevOps functionalities for automating testing and building steps upon pushing new versions.

2.2.4 Comparison

The characteristics of the source code versioning tools presented in the previous sections are summarized and compared in Table 2. In ODIN, Git is selected to be used for source code management. The key characteristics for this decision are its popularity among the consortium

[11] GitHub, https://github.com/ Last access June 2021.

[12] Bitbucket, https://bitbucket.org/product/ Last access June 2021.

[13] GitLab, https://about.gitlab.com/ Last access June 2021.

members, its high speed of operation, its superior branching mechanism facilitating branch-based development workflows, the availability of several popular online repositories (GitHub, Bitbucket, GitLab) and its wide support community.

Table 2: Comparison of source code versioning tools.

	Subversion	Mercurial	Git
Type	Centralized	Distributed	Distributed
Speed	Low	High	High
Branching	Expensive	Expensive	Cheap
Integration in IDEs	✓	✓	✓
Free	✓	✓	✓

2.3 Open Source

At the current stage of the project, it is not yet decided if the developed software will be released, partly or as a whole, open source. This decision will be made during the course of the project, and in coordination with the exploitation activities of WP9.

Until this decision is made, all source code, configuration, documentation and bug report files will be kept private within the consortium. Content will be freely accessed by the members of the consortium or necessary third parties, such as open callers. However, the code should be maintained in such a manner that it is ready for migration to an open source public repository, if this is decided. Details about the type of access to the source code will be also reported in platform deliverables (D3.10, D3.11, D3.12), as well as exploitation deliverables (D9.2, D9.3, D9.4).

2.4 ODIN guidelines

In ODIN, source code management will be handled using Git and a GitLab repository. Developers are free to choose the programming language and the development environment to develop their components. However, they should follow the following guidelines regarding their Git repository. This will ensure consistency across components, ease of use and maintenance of the source code and ease of maintenance of all repositories, e.g., in case of migration to a different host.

- Developers should use Git[14] for source code versioning.
- There should be one Git repository per component. “Component” here means a distinct piece of software performing a distinct set of functionalities, e.g., a web application, a AI module, a set of analytics web services, a robot navigation system. The repository should be as self-contained as possible, so that a developer can clone or fork it and start working with it directly. A repository can be organized into sub-modules, if they are conceptually closely related to each other and to the overall functionality of the component. In this case, the developers can use the git-submodule[15] feature, to manage these sub-modules.
- The Git repositories should be pushed to the ODIN GitLab server, hosted at:

  https://gitlab.odin-smarthospitals.eu
- In case the component needs building for it to execute (e.g., a compiled executable or a web application distribution), the developer is encouraged to use one of the build automation tools described in Section 3.1, depending on the development environment and language used. This will facilitate dependency management and use by other developers. In this case, the developer should include in the repository any configuration files needed by the build automation tool, e.g., Makefiles, package.json files, etc. See Section 3.1 for more information.
- The developers are encouraged to include unit and integration tests in the source code, in a distinct sub-directory within the repository.
- Each repository should contain a file named `Dockerfile`, which describes how to build the component into a Docker image (see Section 3.1.7 for details about Docker). Details about how this `Dockerfile` should be structured are provided in Section 3.3. If more than one `Dockerfiles` are needed to build the component, this could be an indication that the component needs to be split into more than one components (see the second bullet point above). The `Dockerfile` may be generated by build automation tools (see Section 3.1), in which case this should be explained in the `README.MD` file (see below).
- Each repository should contain a file named `README.MD` (preferably all capitalized), containing information about the following:
  - Getting stated / Use: A short description of the component’s functionalities and a short guide for end users to start using the module.

[14] Git, https://git-scm.com/ Last access June 2021.

[15] Git-submodule, https://git-scm.com/book/en/v2/Git-Tools-Submodules Last access June 2021.

  - How to build, Install, deploy: A short guide for developers, maintainers and deployers.
  - Testing (optional): A short guide for developers and maintainers on how to run the developed unit or integration tests.
  - Contributing (optional): A short guide for developers and maintainers on how to contribute to the development of the component, including the code conventions used, the code incorporation process (e.g., pull requests), the different Git branches present, etc.
  - Credits / Getting help (optional): Contact information of the developers/contributors to the component, so that users or other developers can address questions/issues/problems. In case issue and bug tracking is handled by issue tracking services, this part should include references to these services.
  - Licence: A short summary of the licence applying to the module.
- Each repository should contain a file named `LICENCE.TXT` (preferably all capitalized), containing the licence declaration under which the component is distributed (e.g. GPL, MIT, etc.). A short version of this licence should be included in `README.MD`.
- Each repository should contain a file named `NOTICE.TXT` (preferably all capitalized), describing the dependencies, IPR owners, etc. as explained in the ODIN licence policy of D1.2 Data Management Plan and its updated versions.
- When committing code and using source code management tools, it is recommended to follow Git best practices[16], such as the following:
  - Make atomic commits, i.e. one logical change per commit.
  - Do not commit generated, compiled, binary or large files, whenever possible. Properly use the `.gitignore` file to avoid accidental commit of these files. There are many relevant `.gitignore` templates available[17].
  - Do not commit dependencies, use package management or git-submodule[18].
  - Do not commit local configuration such as passwords, or absolute file system references.
  - Write useful commit messages.

[16] Git best practices, https://acompiler.com/git-best-practices/ Last access June 2021.

[17] Gitignore, https://github.com/github/gitignore Last access June 2021.

[18] Git-submodule, https://git-scm.com/book/en/v2/Git-Tools-Submodules Last access June 2021.

- Adhere to the agreed workflow, such as tagging releases, using branch naming conventions and avoiding rewriting history.
- Test before pushing.

Developers should follow common source code **documentation** guidelines and best practices[19], documenting at least the developed APIs (functions, classes, modules, web services). The source code should be written as much clearly as possible, so that minimum documentation is needed, in parts where the meaning of the code is not directly visible. Developers should use existing documentation tools and frameworks available for the programming language and environment they are using, such as Doxygen[20], Javadoc[21], Sphinx[22], etc. All component documentation will be also released as part of the ODIN knowledge base (see Section 9.2), as part of the activities of T3.4.

[19] Google style guide, https://google.github.io/styleguide/docguide/best_practices.html Last access June 2021.

[20] Doxygen, https://www.doxygen.nl/index.html Last access June 2021.

[21] Javadoc, https://www.oracle.com/java/technologies/javase/javadoc-tool.html Last access June 2021.

[22] Sphinx, https://www.sphinx-doc.org/en/master/ Last access June 2021.

3 Building software

In order for the source code to be executed in the operation environment, it needs to be built into an executable program (i.e. binary, webapp, library, etc.). Building is often a complicated procedure, since it involves managing external dependencies, such as libraries and other cooperating components. Build automation tools can facilitate this procedure by providing mechanisms to clearly define the building procedure and manage dependencies. At the same time, containerization tools allow applications to run in any environment, significantly easing the deployment process. This section provides an overview of the build automation and containerization tools that will be used in ODIN.

3.1 Build automation tools

Build automation tools aim to facilitate the compilation of the source code of a project into one or more ready-to-use applications or libraries, such as binary executables, web application distributions, mobile apps, etc. Build automation tools describe the steps needed to build a piece of software, the dependencies needed and their required versions, etc. This section overviews some of the most used build automation tools, each more or less targeting a different programming environment and language.

3.1.1 Make

The utility `make`[23] dates back to 1976 and has traditionally mostly been used for building C/C++ software, especially in Unix-like systems. The specification of the build process is defined in a special file called `Makefile`. Each step of the build process is described by a rule specifying the target object (e.g., an executable file or a library object), its dependencies, i.e., the files that it uses for its construction, (e.g., source code files, header files or other libraries), and the set of commands that act on the dependencies in order to construct the target object. Any available utility, e.g., the `gcc` compiler, can be used in the commands. When the source code is changed, the developer can run the `make` program, which builds the target object by running the specified commands. If the target object already exists, make decides whether it needs to be rebuilt by checking if any of its dependencies has been modified since the previous build. Apart from building, a `Makefile` can also specify other types of actions, such as installing the built application in a particular directory, or cleaning up intermediate files. The specification of `make` is quite generic, so that it is not limited to building executable applications, but any type of file that is constructed out of other files, e.g., images or PDF documents.

An example `Makefile` can be seen below. Each object file to build (`*.o` files) is specified as a target that depends on a number of source and header files. The main executable program,

[23] GNU make, https://www.gnu.org/software/make/ Last access June 2021.

`edit`, is defined as the first target that depends on all the other object files. The order in which rules are written does not matter, since the order is encoded in the dependencies of each target. Targets with no dependencies, such as `clean`, can be used to perform tasks such as cleaning up intermediate files or installing the built executable.

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

3.1.2 CMake

CMake[24] is a cross-platform build automation tool that can be used to compile software in a multitude of development environments and in a compiler independent manner. It is designed in such way allowing to be used with the native build environment. While `make` uses a `Makefile` to compile an executable, CMake operates one level higher and is used to generate `Makefile`s, Ninja build files, KDEvelop, Xcode, or configurations for other types of build systems, such as

[24] CMake, https://cmake.org/ Last access June 2021.

Visual Studio project files. The developer specifies the building process and the dependencies in a platform-agnostic file, called `CMakeList.txt`, which is then translated into the appropriate build system configuration files, according to the language/IDE/operating system used. An example `CMakeList.txt` file is presented below.

```
# CMakeLists files in this project can
# refer to the root source directory of the project
# as ${HELLO_SOURCE_DIR} and
# to the root binary directory of the project
# as${HELLO_BINARY_DIR}.

cmake_minimum_required (VERSION 2.8.11)
project (HELLO)

# Recurse into the "Hello" and "Demo" subdirectories.
# This does not actually cause another cmake executable to
# run. The same process will walk through the project's
# entire directory structure.

add_subdirectory (Hello)
add_subdirectory (Demo)

# Create a library called "Hello" which includes the
# source file "hello.cxx".
# The extension is already found. Any number of sources
# could be listed here.

add_library (Hello hello.cxx)

# Make sure the compiler can find include files for our
# Hello library when other libraries or executables link
# to Hello

target_include_directories (Hello PUBLIC
      ${CMAKE_CURRENT_SOURCE_DIR})

# Add executable called "helloDemo" that is built from the
# source files "demo.cxx" and "demo_b.cxx". The extensions
# are automatically found.

add_executable (helloDemo demo.cxx demo_b.cxx)

# Link the executable to the Hello library. Since the
# Hello library has public include directories we will
# use those link directories when building helloDemo

target_link_libraries (helloDemo LINK_PUBLIC Hello)
```

3.1.3 Catkin

Catkin[25] is the official build system of ROS (Robot Operating System) and the successor to the original ROS build system. It combines CMake macros and Python scripts to enhance the original functionality of CMake's normal workflow. It is designed in such a way to allow better distribution of packages, better cross-compiling support and better portability. Catkin's workflow is very similar to CMake's but adds support for automatic "find package" infrastructure and building multiple, dependent projects at the same time. ROS requires its own custom build system (i.e., Catkin) since there are lots of independent packages which depend on each other, utilize various programming languages, tools, and code organization conventions, within a single ROS project. There are three main types of dependency files that need to be configured for a ROS package within a ROS project:

- `package.xml`: This file is responsible for ordering of the configure step (`cmake`) sequence for catkin-packages in catkin workspaces, define packaging dependencies for bloom (what dependencies to export when creating debian pkgs), define system (non-catkin-pkgs) build dependencies for rosdep, and document build or install or runtime dependency for roswiki / graph tool (rqt_graph). An example `package.xml` file is shown below.

```
<package>
    …
    <name>example_pkg</name>
    <buildtool_depend>catkin</buildtool_depend>
    <build_depend>cpp_common</build_depend>
    <build_depend>log4cxx</build_depend>
    <test_depend>gtest</test_depend>
    …
    <run_depend>cpp_common</run_depend>
    <run_depend>log4cxx</run_depend>
</package>
```

- `CMakeLists.txt`: In general, `CMakeLists.txt` is responsible for preparing and executing the build process using the enhanced functionality of CMake as described above. An example is shown below.

[25] ROS Catkin, http://wiki.ros.org/catkin Last access June 2021.

```
find_package (catkin REQUIRED COMPONENTS cpp_common
          geometry_msgs)
find_package (Log4cxx QUIET)
generate_messages (DEPENDENCIES geometry_msgs)

catkin_ package (
    CATKIN DEPENDS cpp common geometry msgs
        DEPENDS Log4cxx
)

add_library (example_lib src/example.cpp)
target_link_libraries (example_lib
        $(catkin_LIBRARIES) $(LOG4CXX_LIBRARIES))
add_dependencies (examplelib geometry msgs gencpp)
```

- **`setup.py`**: If a package declares Python modules for other packages to use, those need to be declared in a **`setup.py`** file. The names used there could be names of catkin packages or packages distributed over Pypi. An example is shown below.

```
from distutils.core import setup

setup (
    …
    requires = ['rospy']
)
```

3.1.4 Maven

Apache Maven[26] is a system used to build and manage Java-based projects. Maven can simplify and automate the initiation of Java projects, the building process and dependency management. It can be used to automate a several phases of a project's development, including source code validation, compilation, testing, packaging, verification, installation and deployment.

The initiation of a Maven project is based on plugins implementing several *archetypes*, i.e., templates for specific types of projects, such as command-line applications, web applications, plugins, etc. Initiating a project based on an archetype creates the necessary directory structure and configuration files that will be needed for the build process.

[26] Apache Maven, https://maven.apache.org/ Last access June 2021.

The main configuration file for Maven is the `pom.xml` file. It is and XML file specifying the Project Object Model (POM) of the application, containing most of the information needed to build the application and manage its dependencies. An example `pom.xml` file can be seen below[27].

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                 instance"
         xsi:schemaLocation="http://maven.apache.org/
                 POM/4.0.0 http://maven.apache.org/
                 xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>
            1.7
        </maven.compiler.source>
        <maven.compiler.target>
            1.7
        </maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

[27] Example taken from https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html Last access June 2021.

3.1.5 PIP

Pip[28] is a package installer for Python projects. Python is an interpreted language, hence there is no notion of compiling or building source code to an executable. However, pip is included in this list, since it can be used to manage dependencies for Python projects, similar to the dependency management functionalities of build automation tools.

Pip is essentially a packages installer for Python libraries. Developers can use pip to install a Python library, mainly from the Python Package Index[29]. Pip can be run from the command line and performs by downloading the requested package and installing it at the proper location in order to make it available to Python code.

When creating a project involving several external libraries, it is important to list them in a formal manner so that other developers can collectively install all necessary dependencies in order to execute the application. To specify and automate dependency installation, developers can create a `requirements.txt` file including all necessary libraries and their versions. This file can be provided as a command line input to the pip tool, which downloads and installs all of them, respecting the corresponding version numbers and using the latest versions of the required libraries, when this is allowed by the specification. An example `requirements.txt` file can be seen below[30]. Version specifiers[31] can be used to require that a library version exactly matches a required one, is greater than a required minimum, etc.

```
# Requirements without Version Specifiers
nose
nose-cov
beautifulsoup4

# Requirements with Version Specifiers
docopt == 0.6.1         # Must be version 0.6.1
keyring >= 4.1.1        # Minimum version 4.1.1

# Refer to other requirements files
-r other-requirements.txt
```

[28] PIP, https://pypi.org/project/pip/ Last access June 2021.

[29] Python Package Index, https://pypi.org/ Last access June 2021.

[30] Example modified from https://pip.pypa.io/en/stable/cli/pip_install/#example-requirements-file Last access June 2021.

[31] PIP version specifiers, https://www.python.org/dev/peps/pep-0440/#version-specifiers Last access June 2021.

3.1.6 NPM

NPM[32] is the build and packaging system of the Node.js sever and is a popular option for developing web applications. NPM can be used to initiate a project, manage its dependencies and build the source code into a distribution-ready package.

The core configuration for NPM is defined in a JSON file named package.json, which holds information such as the project name, its dependencies, and any scripts to run in order to test and build the application. NPM offers a command line interface that uses `package.json` to create the initial project structure, download the dependencies and manage the available scripts for performing build and other actions. An example package.json file can be seen below[33].

```
{
  "name": "test-project",
  "description": "A test project",
  "main": "src/main.js",
  "scripts": {
    "dev": "webpack-dev-server --inline --progress –
            config build/webpack.dev.conf.js",
    "start": "npm run dev",
    "test": "npm run unit",
    "build": "node build/build.js"
  },
  "dependencies": {
    "vue": "^2.5.2"
  },
  "devDependencies": {
    "autoprefixer": "^7.1.2",
    "babel-core": "^6.22.1",
    "babel-eslint": "^8.2.1",
    "webpack-merge": "^4.1.0"
  },
  "engines": {
    "node": ">= 6.0.0",
    "npm": ">= 3.0.0"
  }
}
```

[32] NPM, https://www.npmjs.com/ Last access June 2021.

[33] Example modified from https://nodejs.dev/learn/the-package-json-guide Last access June 2021.

3.1.7 Gradle

Gradle[34] is a general-purpose build automation tool and is the default build tool used by Android Studio to build Android mobile application projects. The main configuration script for a project is specified in a file usually named `build.gradle`, and includes information such as the project name, SDK versions used, build configuration, dependencies, etc. Gradle scripts can be written in either the Groovy[35] or Kotlin[36] domain-specific languages (DSLs), with Groovy being the one used by Android Studio. Android Studio uses multiple Gradle scripts to specify a single project, describing application-wide or module-wide build configurations. An example Gradle script used for an Android module is shown below[37].

[34] Gradle, https://gradle.org/ Last access June 2021.

[35] Groovy language, https://groovy-lang.org/ Last access June 2021.

[36] Kotlin language, https://kotlinlang.org/ Last access June 2021.

[37] Example modified from https://developer.android.com/studio/build Last access June 2021.

```
apply plugin: 'com.android.application'

/* Android-specific build options. */
android {
    compileSdkVersion 28
    buildToolsVersion "30.0.2"

    defaultConfig {
        applicationId 'com.example.myapp'
        minSdkVersion 15
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
    }

    buildTypes {
        release {
            minifyEnabled true
        }
    }
}

/* Dependencies required to build the module */
dependencies {
    implementation project(":lib")
    implementation 'com.android.support:appcompat-
                    v7:28.0.0'
    implementation fileTree(
        dir: 'libs',
        include: ['*.jar']
    )
}
```

3.1.8 Bazel

Bazel[38] is an open-source build and test tool similar to Make, Maven, and Gradle. It uses a human-readable, high-level build language. Bazel supports projects in multiple languages and builds outputs for multiple platforms. Bazel supports large codebases across multiple

[38] Bazel, https://bazel.build/ Last access June 2021.

repositories, and large numbers of users. It uses **`BUILD`** files written in Starlark[39] in order to direct Bazel on what to build and how to build it. A build target specifies a set of input artefacts that Bazel will build plus their dependencies, the build rule Bazel will use to build it, and options that configure the build rule. A build rule specifies the build tools Bazel will use, such as compilers and linkers, and their configurations. Bazel ships with a number of build rules covering the most common artefact types in the supported languages on supported platforms.

The process when running Bazel is as follows. First the **`BUILD`** files relevant to the target are loaded. Then these files are being analysed to check their inputs and dependencies. The build rules are applied and the action graph is produced. The action graph represents the build artefacts, the relationships between them, and the build actions that Bazel will perform. Thanks to this graph, Bazel can track changes to file content as well as changes to actions, such as build or test commands, and know what build work has previously been done. The graph also enables you to easily trace dependencies in your code. Finally the build actions are executed on the inputs until the final build outputs are produced.

Below is an example of two **`BUILD`** files in a project tree where the second one has a dependency on the first.

```
cc_library (
    name = "hello-time",
    srcs = ["hello-time.cc"],
    hdrs = ["hello-time.h"],
    visibility = ["//main:__pkg__"],
)
```

[39] Starlark language, https://docs.bazel.build/versions/4.1.0/skylark/language.html Last access June 2021.

```
cc_library (
     name = "hello-greet",
     srcs = ["hello-greet.cc"],
     hdrs = ["hello-greet.h"],
)

cc_binary (
     name = "hello-world",
     srcs = ["hello-world.cc"],
     deps = [
         ":hello-greet",
         "//lib:hello-time",
     ]
)
```

3.2 Containerization

The output of the building process is an executable file that can be run in the execution environment, e.g., in a hospital's server. However, this is not the end of the story, especially for large applications. In most cases, apart from the executable file(s), one needs to setup the environment in which to run the application: install the necessary libraries, utility programs, servers, etc., on which the application depends. As an example, an application predicting future hospital needs based on monitored history may consist of a web interface and a backend written in Python. In order for this to run on a target machine, one would need to setup a web server to server the web interface, install Python for the backend, install Python libraries to support the communication between the web interface and the backend, install machine-learning Python libraries used by the prediction mechanism, etc.

Containerization software facilitate the execution of an application in a target machine by allowing software to run in an isolated environment, complete with all dependencies needed. The goal is that once the complete container is available, it is all that is needed (apart from the containerization software itself) to run the application in any target machine.

The most prominent containerization software currently in use is Docker[40]. Containerization in Docker is based on the key concepts of images and containers. An *image* is a miniature file system containing the necessary files and directory structure needed for an application to run. A developer can create an image by using an existing image, e.g., of a Linux distribution, and by adding to it additional layers of files, e.g., installing a web server, or copying files from the

[40] Docker, https://www.docker.com/ Last access June 2021.

developer's computer. A *container* is an isolated environment created from an image. Users can start a container and run applications inside it, knowing that they will be run in the isolated environment specified in the image. The container can optionally communicate with other containers or with the host machine.

In order for a container to use data stored in the host machine, or in order for multiple containers to share data among them, Docker uses *volumes*. Volumes are Docker's mechanism for persistent storage. They can be created outside the context of any container, mounted to specific filesystem directories and used by containers to read or write data.

The specification of a Docker image is written in a `Dockerfile`. There is extensive documentation available on how to write a `Dockerfile`[41]. A `Dockerfile` contains instructions on how to create the isolated environment to be used in a container. As an example, consider the following `Dockerfile`, used to build a Node.js web application[42].

```
FROM node:14

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json ./
RUN npm install

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

Each line consists of a Docker command, such as `FROM`, `WORKDIR` and `COPY`. The first line gets an existing Docker image as a starting point for this image. The `node:14` image is a minimal Linux distribution including only the files necessary to run a Node server. The `WORKDIR` command sets the current working directory within the image's filesystem. The `COPY` command copies the `package.json` file (see Section 3.1.6 for details about `package.json`) from the current directory within the host's filesystem to the working directory of the image. Then the `npm`

[41] Dockerfile description, https://docs.docker.com/engine/reference/builder/ Last access June 2021.

[42] Example modified from https://nodejs.org/en/docs/guides/nodejs-docker-webapp/ Last access June 2021.

`install` command is run inside the image to install the necessary dependencies, as described in `package.json`. The code of the application is then copied to the working directory of the image, so that all relevant files for the webapp are now available in the image. The final two commands are instructions used whenever a container is executed using this image, that tell how to run the application: which port to expose and which command to run for running the application.

3.3 ODIN guidelines

In ODIN, the developers should use a build automation tool such as the ones described in Section 3.1. Using such a tool allows the formal specification of the build process in the appropriate configuration files, which will be submitted to version control along with the source code of the application, and consequently its automation. The developers are free to choose the build automation tool that is most suitable for their development.

In ODIN, each software component will be wrapped in a Docker image that can be used to run the component at a specific container within the target deployment. To achieve this, the developer needs to specify two types of files:

- The build automation description file, e.g., a `Makefile`, a `package.json` file, etc., describing the steps needed in order to build the target object, e.g., an executable, a library, a website, etc., meant to be used by a build automation tool, such as CMake, NPM or Gradle. This file (or files) needs to be present in the Git repository of the software component, along with the source code.
- A `Dockerfile`, organized in two stages, when possible:
    - Stage 1: How to use the selected build automation tool to build the component out of its source code.
    - Stage 2: How to create the Docker image containing the built component.

As a concrete example, the following Dockerfile[43] describes how to build a Docker image of an Angular web application, organized in two stages.

[43] Example modified from https://dzone.com/articles/how-to-dockerize-angular-app Last access June 2021.

```
# Stage 1
# ------------------------------------------------------
FROM node:10-alpine as build-step

RUN mkdir -p /app
WORKDIR /app
COPY package.json /app

RUN npm install
COPY . /app

RUN npm run build --prod


# Stage 2
# ------------------------------------------------------
FROM nginx:1.17.1-alpine

COPY --from=build-step /app/docs /usr/share/nginx/html
```

This `Dockerfile` essentially describes two images, one used to create the built application and the other being the main image of the component. In Stage 1, the production-level website is built using the NPM build automation tool and the associated `package.json` file. A key difference with the example of Section 3.1.7 is the `as build-step` part of the first line, which specifies an alias, "build-step" for the first stage image. This image will only be a temporary image, to support the creation of the main image of Stage 2. The following lines until Stage 2 are used to copy the necessary files in this temporary image and run the appropriate commands to build the production website (see Section 3.1 for build tools). The final files of interest are stored in the `/app/docs` directory inside the `build-step` image.

In Stage 2, the produced website is copied to a fresh image, which is the main image of the component, and the one that will be used to run containers. The main thing to notice is the `–from=build-step` parameter in the `COPY` command, which instructs Docker to copy files from another image (the one aliased `build-step`) to the current working image. Note that the images of the two steps need not start from the same base image. They are quite independent in how they are constructed.

The above described two-stage approach permits the complete description of how to create a runnable Docker image in a single `Dockerfile`. The developer does not need to first build the executable and then create a Docker image; creating the Docker image takes care of buiding the software as well. At the same time, the run-time execution environment is separated from the build-time execution environment, since they are both containerized in separate images. However, only the result of the second stage persists and constitutes the output of the containerization process. This 2nd stage image contains the executable files that are used by the end-user in a container. The two-stage approach can also be extended to more steps, if more processes are required in order to build the executables or the Docker image.

4 Testing software

Software testing is a crucial part of the development process prior to release. Proper and frequent testing ensures the robustness of the developed applications and early captures issues inadvertently introduced when new functionality is added.

Four levels of software testing are commonly recognized, as depicted in Figure 5:

- Unit testing: Testing individual units of code in isolation to assess if they fulfil their desired functionality.
- Integration testing: Testing the interconnection of components in a larger part of a system, to assess how they operate in combination.
- System testing or Developers Acceptance Testing (DAT): Testing the entire system as a whole.
- Validation: Broader evaluation of the final phases of development, to ensure that functional and non-functional requirements are met, and to evaluate user acceptance.

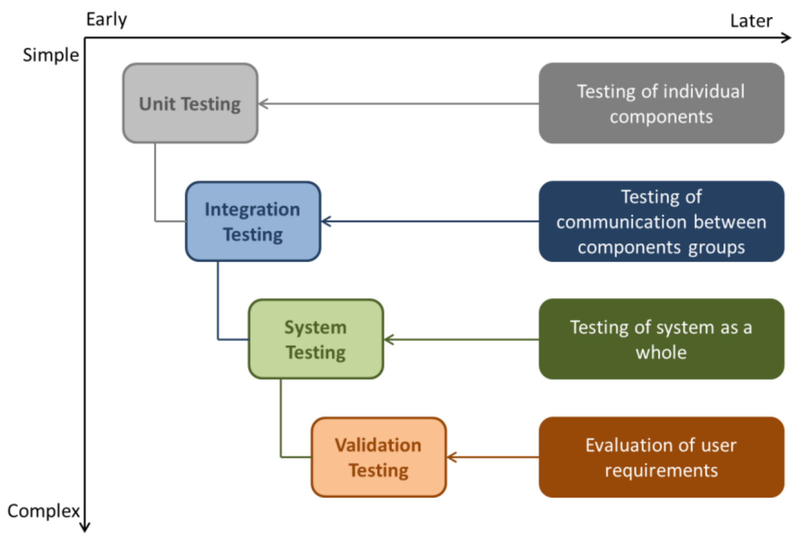

Early
Later
Simple
Complex
Unit Testing
Testing of individual components
Integration Testing
Testing of communication between components groups
System Testing
Testing of system as a whole
Validation Testing
Evaluation of user requirements

Figure 5: Software testing levels.

The "system testing" and "validation" levels involve the entire system and evaluate its compliance to functional and non-functional requirements. These types of testing will be performed as part of the activities of WP7 "ODIN Pilots Design, Deployment, Evaluation and Validation". In the context of WP3 and this deliverable, the relevant testing levels are "unit testing" and "integration testing", which test the functionality of individual components, their APIs (Application Programming Interfaces) and their interconnection. Such kinds of tests can be well-defined enough to be automated and included in the automated DevOps pipeline.

4.1 Unit tests

Unit testing refers to testing individual *units* of code, such as functions, classes, or modules, in terms of whether they meet their design requirements. Units are usually small parts of an application that perform a single task or a small set of tasks. Unit tests evaluate the performance of these blocks of code in isolation, by providing them with example input and comparing their output to the expected output.

Unit tests are implemented as additional code written by the developer of the application along with the code of the main functionality. The unit test is an isolated piece of code that uses a single unit, e.g. a function, including only the necessary code to use the unit. The test code provides the unit with input that ideally covers the whole range of input that can be provided to the unit, including extreme cases or erroneous input, runs the unit and reports the results, possibly comparing them to the expected output in each case.

Writing and updating unit tests as the code progresses adds an additional burden to the developer, since they require time and more thinking. However, the benefits significantly outweigh this burden in the short- and long-run, since unit tests lead to easier maintenance.

Adding and running unit tests in an application at the time of development has several advantages:

- They can detect bugs early during code development. Resolving bugs in small pieces of code is easier than trying to resolve the same bugs after the whole system is implemented.
- The process of writing the unit tests forces the programmer to think more carefully about the unit being developed, of its possible inputs and outputs. This can lead to a clearer implementation of the unit's functionalities, an in turn to easier maintenance of the project.
- The existence of unit tests allows the programmer to be more confident about modifications in the code. After making a change in the code, the programmer can run all unit tests automatically to see if the change has unexpectedly influenced another component in the system.

Depending on the programming language used, there are several frameworks or libraries that can be used to assist the developer in defining and running unit tests. Although the programmer could manually write the tests, if desired, these frameworks reduce the time needed to construct the unit tests, making it easier to create unit tests together with the functional code. Some popular choices for common programming languages and environments are the following:

- C/C++: Catch2[44], GoogleTest[45], Boost.Test[46], NUnit[47], Visual Studio native C++ unit tests[48]

[44] Catch2, https://github.com/catchorg/Catch2 Last access June 2021.

[45] GoogleTest, https://github.com/google/googletest Last access June 2021.

[46] Boost.test, https://www.boost.org/doc/libs/1_75_0/libs/test/doc/html/index.html Last access June 2021.

[47] NUnit, https://nunit.org/ Last access June 2021.

- Java: JUnit[49], TestNG[50], JBehave[51]
- Python: unittest[52], pytest[53]
- ROS: rostest[54], GoogleTest, Python unittest
- JavaScript: Mocha[55], Jest[56], Chai[57], Jasmine[58]
- R: testthat[59]

4.2 Integration tests

While unit tests test isolated units of an application, integration tests test the connection between different parts of an application, or between applications in a larger system. Integration tests are used to evaluate the performance of large components and their APIs (Application Programming Interfaces) in terms of their compliance to the functional requirements. Integration tests are more complex to define than unit tests and they may be more difficult to evaluate, since they involve larger workflows involving several components.

Integration testing involves the detailed definition of a test case scenario, its execution, either automatically or manually, and the reporting of the results. Two common types of integration testing are big-bang testing and bottom-up testing. In big-bang testing, the components are combined to form the application of interest and then the combined system is used for integration testing. In bottom-up testing, the low-level pieces of an application are tested first (e.g. in unit tests), and are then combined to form larger structures in the hierarchy. Tests are performed iteratively at each level of the hierarchy until the whole application is tested.

[48] Visual Studio native C++ tests, https://docs.microsoft.com/en-us/visualstudio/test/getting-started-with-unit-testing?view=vs-2019&tabs=mstest Last access June 2021.

[49] JUnit, https://junit.org/junit5/ Last access June 2021.

[50] TestNG, https://testng.org/doc/ Last access June 2021.

[51] JBehave, https://jbehave.org/ Last access June 2021.

[52] Python unittest, https://docs.python.org/3/library/unittest.html Last access June 2021.

[53] Python pytest, https://docs.pytest.org/en/6.2.x/ Last access June 2021.

[54] ROS rostest, http://wiki.ros.org/rostest Last access June 2021.

[55] Mocha, https://mochajs.org/ Last access June 2021.

[56] Jest, https://jestjs.io/ Last access June 2021.

[57] Chai, https://www.chaijs.com/ Last access June 2021.

[58] Jasmine, https://jasmine.github.io/ Last access June 2021.

[59] R testthat package, https://testthat.r-lib.org/ Last access June 2021.

Depending on the complexity of an integration test scenario, it may be executed either automatically or manually. In automatic testing, the test scenario is written in a formal manner, e.g. as a Jenkins pipeline (see Section 8.1.1 for a description of Jenkins), which can then be run automatically by a testing framework. In manual testing, the individual steps of a test scenario are written in detail and are then performed by a human operator. At each step, the expected output is specified, so that the tester can report the success or failure of each step. Manual testing systems such as Squash[60] can be used to facilitate the definition of test case scenarios, and reporting the test results.

4.3 Test server infrastructure

A test infrastructure will be provided to facilitate the task of testing the software components prior to production deployment, in an environment intended to resemble operating conditions that will be present in the actual pilot sites.

The test infrastructure will be provisioned in the cloud infrastructure that Inetum has at its datacentre located in Murcia, Spain. Table 3 provides a brief description of its design and performance features.

[60] Squash, https://www.squashtest.com/?lang=en Last access June 2021.

Table 3: Test environment cloud datacenter features.

Scope	Features
Facility	• Tier IV. 99.999% availability. • Anti-seismic construction with insulated electromagnetic. • Redundant infrastructure for mission environments review. • 1.2 MW of power maximum in datacentre of high density. • Double electric ring with 2 UPS and 2 groups generators with 1-week autonomy. • GREEN IT "Base Design", being 50% more efficient in the consumption of energy.
Security	• Specialized security personnel 24x7. • Intelligent indoor and outdoor video surveillance system with intruder detection. • Access to critical rooms controlled by facial biometrics (TI, MPOE, SOC, NOC, etc.). • Very Early Smoke Detection Air (VESDA). • Water extinguishing system mist, avoiding the evacuation of the datacentre.
Operation	• Customized 24x7 support backed by technical team of experts with presence onsite IT and Industrial staff. • ITIL, SSAE16, ISO, ICREA standards level 5.
Communication	• 2N end-to-end redundancy. • Two independent links with diversified access and connection to two neutral points (Telvent and Interxion). • Own public address, balanced between the two links • Safety equipment Service Provider logic, with protection against DDOS attacks. • 2 Multi-Carrier zones (MPOE) for service providers with exterior fingerprint access.

The cloud test infrastructure for the ODIN Project uses private IP addresses to communicate inside the cloud, and public IP addresses to communicate over the Internet. Public IP addresses allow for secure communication from origins to prevent unauthorized access, and each deployed instance has allocated one public IP. The cloud implements firewall protection that allows for the definition of rules to restrict access from specific sites and to specific resources inside the virtual space allocated for the project.

4.4 ODIN guidelines

The developers of ODIN components are encouraged to include **unit tests** in their components, update them as code progresses and new use cases are added, and execute them when making modifications to ensure that no bugs are introduced to existing code. The developers are free to choose the most appropriate unit testing framework for their applications, corresponding to the programming language and environment used.

Each project partner will be responsible for unit testing their own modules and components. Unit tests should at least be run prior to the release of a component (see Section 5). After major changes are made in a component, unit tests should be executed to ensure that the functionality is as expected and that no other parts of the component are damaged. Whenever possible, unit tests should be included as part of the automated DevOps pipeline, in the Jenkins scripts used for orchestration (see Section 8.2 for guidelines regarding Jenkins-based orchestration), so that their execution is triggered upon pushing code to the GitLab server.

After all components of a sub-system of the ODIN platform are implemented, **integration tests** will be performed to ensure that the combined components operate as expected. Integration tests should be automated when possible, or manually defined otherwise.

Automated integration tests should be written as **Jenkins pipelines**, so that they can be automatically triggered upon changes in the source code of the components. A Jenkins pipeline can define the steps to take to perform the integration test, failing upon failure of a specific step. Pipelines can be blocked using the `input` Jenkins step when confirmation from a human tester is needed before resuming the pipeline. The test results can be stored and reported by the pipeline itself, by making use of the “post” blocks of the pipeline script.

Manual integration tests should be written in cases where automated tests are difficult to define. For each integration test, a detailed description of the test scenario needs to be specified, detailing the steps to take and the expected output of each step. The tests should be executed by a human tester and the result of each step recorded. Testing platforms such as Squash[61] can be used for this purpose. The decision to use such platforms will be taken later during the course of the project.

The initial execution environment of the integration tests will be the **ODIN testing infrastructure** provided by Inetum and described in Section 4.3. A detailed design of the services and provided by the testing infrastructure will be provided as part of the deliverables of WP7, when application requirements are specified. Details about how to access the testing server, including URLs and authentication, will be provided to the project partners at the time the testing server is setup. After the integration tests are run successfully in the testing infrastructure, they will be transferred to the actual pilot sites and robotic applications.

Each partner is responsible to write integration tests whenever their components use other components of the same or other partners. Sample input data for the execution of the integration tests can be provided by the component owners or the integration team. The DevOps and integration team is responsible for the execution of the integration tests, either by setting up and the appropriate Jenkins pipelines, together with component providers, and making sure they are correctly triggered and executed, or by running the manual tests and reporting the results.

[61] Squash, https://www.squashtest.com/?lang=en Last access June 2021.

5 Software release

Once a piece of software has been built and tested, it is ready to be released so that it is available for use by end-users. This section describes the types of releases foreseen for the ODIN components, as well as the repositories that will be used to publish released software.

5.1 Software release versions

According to ISO/IEC/IEEE 12207:2017[62], in a typical software development lifecycle, the software passes through specific stages of development, according to its maturity and readiness. The most typical stages of development are the following.

- Pre-alpha: This is the stage of initial software development and unit testing, before formal testing by designated testing procedures.
- Alpha: In this stage, developers test the functionality of the software through a series of white-box, black-box or gray-box techniques, in order ensure that all features are present and to address major bugs. Software released at this stage may be erroneous and unstable.
- Beta: In this stage, software is tested prior to being released to the general public. Tests at this stage are focused on customer acceptance, including usability tests. Software at this stage is used for demonstrations to the general public, but may still be unstable.
- Stable release: In this stage, the software is ready to be released to the general public and is in a fully functional, stable state. This version may also be digitally signed, to guarantee its integrity to customers.

Once software is released to the public, it is still under constant testing and reviewing, either by the public through bug reporting systems, or by the developers themselves, through bug detection and re-designing of functionalities. Further versions of the software pass through the above stages prior to public release.

5.2 Tagging versions

Tagging software versions is important in order to keep track of changes through the various releases. Proper tagging permits dependency handling tools, such as the ones described in Section 3.1, to install the proper versions while building software from source or creating Docker images.

[62] ISO/IEC/IEEE 12207:2017. Standards catalogue. International Organization for Standardization. November 2017. Available at: https://www.iso.org/standard/63712.html. Retrieved 16 June 2021.

Semantic versioning[63] is a principled approach to release tagging, where the version numbers have specific meaning useful to the developer and to automated tools. In semantic versioning, a software release is tagged with a version number of the following format:

```
MAJOR.MINOR.PATCH
```

e.g., 1.4.2, optionally followed by pre-release tags appended after the patch number, separated by a dash, e.g. 1.0.0-alpha.

The components of the tag incremented with the following rules:

- The `MAJOR` version number is incremented when there is a change in the public API exposed by the software.
- The `MINOR` version number is incremented when new functionality is added, in a backwards compatible manner, without affecting the existing exposed API.
- The `PATCH` version number is incremented when backwards compatible bug fixes are made.

After a software version is released with its tag, it must not be altered in any way. Any changes to the software must be released as a new version. This prevents problems with dependencies, since, once a particular version is used by an application, it stays the same at all times.

For initial development, prior to the first release, the `MAJOR` version number should be 0, e.g. version 0.1.1. Once the first software release is made, and the software's API is defined, the `MAJOR` number takes the value of 1 and is then incremented only whenever an API change is made.

5.3 Docker registry

Docker provides the ability to create a registry of Docker images, for dockerized software that is ready to be released. The Docker registry[64] is a service for storing and delivering built images, available in different versions specified by tags. The user can upload and download images using push and pull commands from the command line.

By default, Docker registry uses the host machine's filesystem for storing the images. However, cloud-based systems can also be used for large deployments, such as Amazon S3 buckets, Microsoft Azure, etc. Docker registry handles user authentication through TLS and basic authentication. Docker registry can be configured to provide notifications to the developers in response to events that happen in the registry, such as new available versions.

[63] Semantic versioning, https://semver.org/ Last access June 2021.

[64] Docker registry, https://docs.docker.com/registry/ Last access June 2021.

Docker registry can be used to setup a private registry, with limited access to authorized developers. For larger projects, a public registry can be setup. Docker Hub[65] is such a public service, supporting all functionalities of the Docker registry, but allowing public access to the images, facilitating the development of large-scale software projects.

5.4 ODIN guidelines

During the development of a software component, there will be at least two types of software release: Intermediate and stable. **Intermediate releases** cover the pre-alpha, alpha, and beta phases described in Section 5.1 and may contain new added functionality, small changes, bug fixes, improvements, etc. **Stable releases** are intended to be fully tested versions of the components, ready to be publicly released and used by other developers, and aligned with the overall ODIN platform version.

Regarding the overall ODIN platform, three major releases are expected during the course of the project, expected in M12, M24 and M36, respectively. Each version will include specific components, reflected in the ones available in the Docker registry, and used in the pilot sites.

The released components should follow the **Semantic Versioning** approach described in Section 5.2. In order to avoid frequent changes of the MAJOR version number, it is advised that the API exposed by each component is thoroughly designed during each phase of development, so that it covers all the foreseen cases of interaction.

The released components will in general have their own lifecycle, each maintaining its own versioning numbers, according to their status. This numbering may be different than the numbering used for the overall ODIN platform. A component may in parallel be implementing a new feature in a separate branch while some fix is made in the master branch (see the GitFlow workflow paradigm in Section 2.2.3.1). A particular ODIN platform release may consist of components of various versions, with the restriction that these versions are stable releases of the components.

Each ODIN platform release will be accompanied by the full list of the components it consists of, along with their stable version numbers. This list will be essentially formalized in the docker-compose files (or other similar composition files, e.g., Kubernetes YAML files) that will specify how to deploy the platform's components. As described below in Section 6.3, these files will be published and managed in the GitLab server similar to the source code of the components, but in a different repository, since they describe deployments of composite applications instead of single components. The stable releases of the ODIN platform will only consist of stable releases of their components.

[65] Docker Hub, https://hub.docker.com/ Last access June 2021.

In ODIN, a **private Docker registry** will be used for releasing components. The Docker registry will be available at:

https://registry.odin-smarthospitals.eu

In the initial phases of the project, the registry will be kept private with access provided only to the consortium members. A migration to a public registry, such as Docker Hub, may be considered during the course of the project, following the decisions regarding the platform's exploitation in the corresponding work packages.

Access to the Docker registry will be provided to all consortium members in the first months of the project. User authentication will be handled through the mechanisms described in Section 9.

6 Deployment

Deployment of an application means its installation and use at the end-user environment, which for the ODIN project is the computing infrastructure of the pilot sites (hospitals). Deployment involves selecting and composing the necessary published components and services in order to perform a task of interest. This section describes the tools that will be used in ODIN for composing released components into complete applications, and for automatically updating them once new versions are available.

6.1 Service composition

Service composition deals with composing software modules into large applications. When it comes to software modularity, there are two major approaches:

- Monolithic applications
- Service-oriented software

In monolithic applications, the entire business logic is contained in a single application, which is mostly independent from other applications. Monolithic applications are usually written in a single language or software development framework and usually result in a large codebase with the source code for the entire application. Modularity in monolithic applications is achieved in the source code, employing the language's mechanisms such as functions and classes.

On the other hand, in service-oriented applications, the business logic is split into a number of small applications that communicate with each other through well-defined APIs (Application Programming Interfaces). Each service, also referred to as micro-service, is responsible for a well-defined subset of the complete application logic. It may be written in its own language or framework and maintained by a different set of developers. The logic of the complete application is achieved by composing these software modules.

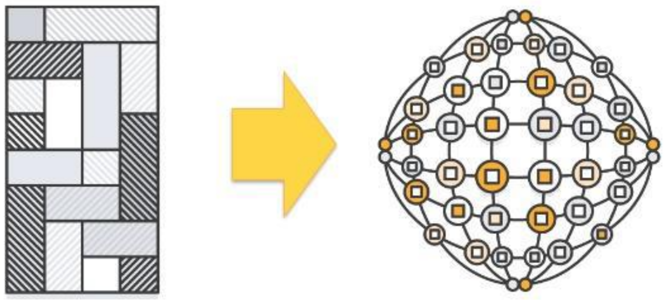


Figure 6: Monolithic vs. micro-service applications.

There are advantages and disadvantages in both approaches. Monolithic applications are easy to deploy and test, since they consist of a single software component. However, they are difficult to scale up. Adding new features and fixing bugs requires altering the whole application, and it is likely that changes in one part will affect other parts of the application as well. Due to this fact, maintaining and updating the software is harder and more time consuming.

The service-oriented architecture was created to address this scalability issue. Each service is self-contained and quite independent from the others, so adding new functionality or resolving bugs is faster and less likely to cause problems in other parts of the whole application. If a service's API is well-defined and respected, the application behind it could change altogether or replaced by another, without affecting other services using the API. However, micro-service applications are more difficult to deploy. Instead of deploying a single application, one needs to deploy a set of applications and manage their communication through proper configuration files.

Micro-service-based applications have one more advantage compared to monolithic applications. Creating small applications that perform a single task well is beneficial in software development in general. One can focus on how to perform the task robustly and fast, without needing to deal with other irrelevant tasks. The developer is encouraged to split the logic into small pieces which are easy to reason about. This makes code clearer, and easier to modify by other developers or the same developer in the future. The resulting pieces of software are reusable and can be combined with other pieces in ways that were not originally planned.

The composition of micro-services requires systems that manage how the services are connected. These orchestrators manage which services should start/stop at any time, how they can be discovered by other components, which ports they expose, etc. This is an extra level of management (and complexity) added by the micro-service architecture, but it allows the creation of highly scalable applications that are easy to maintain by diverse teams of developers.

The containerization functionalities offered by Docker significantly facilitate the creation of micro-service-based applications. Each service can be shipped in its own Docker container, self-contained with any dependencies of the system in which it was developed, and ready to be integrated with other containerized components. On top of these self-contained services, tools for service composition operate to manage their interconnection in large applications.

6.2 Tools for service composition

This section briefly discusses the most popular tools used for the composition of Docker containers in large applications.

6.2.1 Docker-compose

Docker-compose[66] is the most direct way to compose a number of Docker images to create a larger application. Docker-compose is already available in a Docker installation. Docker-compose reads the description of a composition from a designated file usually named `docker-compose.yml`. The `docker-compose.yml` file, written in the YAML syntax, describes which Docker images to use, which ports they expose, how to setup the environment for each service, etc.

As an illustrative example, consider the following docker-compose file, used to deploy a web application[67].

[66] Docker-compose, https://docs.docker.com/compose/ Last access June 2021.

[67] Example modified from https://github.com/compose-spec/compose-spec/blob/master/spec.md Last access June 2021.

```
version: "3.9"

services:
  frontend:
    image: awesome/webapp
    ports:
      - "443:8043"

  backend:
    image: awesome/database
    volumes:
      - db-data:/etc/data
```

It specifies two services, named “frontend” and “backend”. For each service, it specifies which Docker image to use, as well as additional information needed for its deployment. For the front-end service, the “awesome/webapp” image is used, which the ports to use are also setup. For the back-end, the “awesome/database” image is used to setup the application’s database, with an additional configuration for the data volumes to use. Each service can refer to other services by using the names specified in the docker-compose file.

The `docker-compose.yml` file is parsed by the docker-compose engine, which pulls the necessary images from the host machine or the remote repositories, and creates the necessary environments for them to run and interact. The end result is that a set of Docker containers are executed and communicate with each other. In the example above, two servers would execute, one holding the backend database and the other holding the webapp UI, which accesses this database.

Docker-compose supports a rich set of configuration options allowing to design a wide variety of workflows.

6.2.2 Docker swarm

Docker swarm is a group of docker applications that are joined together in a cluster. Docker swarm[68] is therefore a mode of operating the Docker-compose engine described above in Section 6.2.1, so that the complete application can be executed in a distributed manner, e.g. in a computer cluster. The configuration of Docker-swarm is based on `docker-compose.yml` files, similar to docker-compose. However, Docker swarm facilitates cluster deployment, by

[68] Docker swarm, https://docs.docker.com/engine/swarm/ Last access June 2021.

offering orchestrated monitoring, resource allocation, load balancing, etc. The fact that in a Docker swarm there is one manager node and several worked nodes is the reason for achieving high quality resource management and cluster efficiency.

When running in Docker in swarm mode, the user can deploy several services as a *stack*. A stack is defined in a `docker-compose.yml` file, similar to the ones used by docker-compose. When deploying a stack, the user can monitor the running services and start/stop individual services as needed, or stop the entire stack at once.

Some of the most valuable features[69] of a Docker swarm are: a) decentralized access, b) increased security (because of the nature of intra-node communications), c) load balancing, d) scalability and e) roll-back in case the orchestrators need to revert their changes in a previous safe environment.

6.2.3 Kubernetes

Kubernetes is a platform for managing containerized workloads and services[70]. It is portable, extensible and open-source, thus providing a good way to bundle and run large-scale applications. In a production environment, the orchestrator needs to manage the containers that run the applications and ensure that there is no downtime. Kubernetes is able to provide: a) service discovery and load balancing (containers are exposed using IP addresses), b) managing and mounting storage systems, c) rollouts and rollbacks (automate change of container states), d) resource management of containers (CPU and RAM), e) respawning containers that fail and e) storing security information as secrets.

Kubernetes is organized in Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run pods. Nodes contain information regarding the addresses that containers use to communicate, the condition of pods and the capacity. An example of a node specification is shown below.

[69] Docker swarm features, https://www.simplilearn.com/tutorials/docker-tutorial/docker-swarm Last access June 2021.

[70] Kubernetes, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes Last access June 2021.

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

As mentioned previously, Kubernetes runs your workload by placing containers into Pods to run on Nodes. Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. To be more specific, they are a group of one or more containers, which share storage and network resources. Pods are organized in templates, such as the one below.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c',
              'echo "Hello, Kubernetes!" && sleep 3600']
      restartPolicy: OnFailure
```

6.2.4 Amazon EC2 Container Service (ECS)

Amazon Elastic Container Service (Amazon ECS)[71] is a container orchestration service providing easy deployment, management and scaling up of large applications. The containers

[71] Amazon ECS, https://aws.amazon.com/ecs/ Last access June 2021.

are deployed in an AWS (Amazon Web Services) cloud cluster, allowing applications to scale up easily. It adopts a serverless architecture, where the computing resources are automatically allocated, so that the user does not need to deal with server configuration, reducing thus the time needed for setting up a deployment. Amazon ECS provides a free trial, with full-features available according to the available pricing schemes.

6.2.5 Apache MESOS

Apache MESOS[72] is a resource allocation and orchestration system for running applications on computing clusters. MESOS allocates resources to distributed execution frameworks such as Hadoop[73] and MPI[74]. The core part of MESOS is the master daemon, which manages distributed agent daemons deployed on cluster nodes, on which tasks are run according to the MESOS framework used. The master daemon decides the number of computational resources to offer to each framework, according to the selected allocation policy (e.g., fair sharing, strict priority, etc.). The scheduler of the framework running the application can accept the offer and select which of the resources to allocate to the tasks to run, passing this allocation to MESOS, which executes the tasks in the selected agents. MESOS can be used to run containerized applications, using either Docker containers or containers of MESOS's own containerization system.

6.2.6 Nomad

Nomad[75] is an orchestration system that focuses on cluster management and scheduling, aiming to reduce the complexity added with other types of features such as service discovery and monitoring offered by other orchestrator systems, such as Kubernetes. Nomad can scale up to thousands or millions of nodes and can run not only containerized applications, but also virtualized and standalone ones.

6.2.7 Comparison

The characteristics of the service composition tools presented in the previous sections are summarized and compared in Table 4. For ODIN, Docker-compose and Docker swarm are selected for container orchestration, with Kubernetes also being an alternative that can be setup during the course of the project, if requirement analysis reveals that its functionalities cover better the needs of hospital deployments. The selection of these tools is based on the combination of free distribution, ease of setup and configuration, ability to scale to computing clusters and wide community. The ease of setup of Docker-compose and Docker swarm make

[72] Apache MESOS, http://mesos.apache.org/ Last access June 2021.

[73] Apache Hadoop, https://hadoop.apache.org/ Last access June 2021.

[74] OpenMPI, https://www.open-mpi.org/ Last access June 2021.

[75] Nomad, https://www.nomadproject.io/ Last access June 2021.

them the first choice considered during the early phases of the project. Familiarity of the consortium members and the DevOps team with these tools is also a reason for their selection.

Table 4: Comparison of service composition tools.

	Docker-compose	Docker swarm	Kubernetes	Amazon ECS	MESOS	Nomad
Free	yes	yes	yes	no	yes	yes
Ease of setup	high	high	low	high	low	high
Resource allocation	no	yes	yes	yes	yes	yes
Scalability	low	medium	high	high	high	high
Built-in features	low	medium	high	high	medium	medium

6.3 ODIN guidelines

In ODIN, we will use **docker-compose** for deploying components in the target environment. For each deployable system of ODIN, the developers should create a `docker-compose.yml` file that describes how to compose the system from individual components, e.g., a web application by composing a server, a database and a GUI. The docker-compose.yml file will specify all needed configuration for an application, which may include the Docker images to use, ports to make available, environment variables, etc.

The `docker-compose.yml` file will be uploaded to the project's **GitLab** server, at https://gitlab.odin-smarthospitals.eu, and submitted to version control, similar to the source code of the components. However, the `docker-compose.yml` file will be stored in a different repository than the one used for the source code. In general, there will not be an one-to-one correspondence between a component and a deployable system, since the system may be composed of multiple components, hence the `docker-compose.yml` will not correspond to a single source code repository. For this reason, there will be a separate repository where all `docker-compose.yml` files will be stored. The docker-compose files in this repository will also serve as a registry for all deployable services in ODIN.

To coordinate applications running in multiple computing nodes, **Docker swarms** will be used, at least at the initial phases of the project. Composite applications will be deployed as Docker stacks, specified in the corresponding docker-compose.yml files. The services in the deployed stacks will be monitored either through the command line interface, or through the Graphical User Interface, as described in Section 7.4.

If during the course of the project, and through the analysis of the system and end-user requirements, it is decided that Docker swarms are not adequate for the criticality of the hospital pilot site environment, container composition and orchestration may be switched to **Kubernetes**, which offers increased resilience for critical large-scale applications. This decision will be taken within the activities of T3.1, through coordination among the project's technical partners and any decisions will be recorded in relevant deliverables of WP4 or WP7.

7 Operation monitoring and feedback collection

During the operation phase, an application is being used by the end-users performing its specified functionality. The feedback collected by the end-users during this phase regarding the operation of the software and any problems and issues that may arise from its usage is very important for developers. Using this feedback, they can solve problematic behaviour, or alter the functionality of components in order to better satisfy user requirements.

Operation monitoring refers to the ability to manage a deployment, i.e., have an overview of the services that are running in the deployment, start/stop services as needed, view service logs, update services to newer versions, etc. This type of monitoring can be achieved either through the command line interface of the container orchestration tools (Docker swarms, Kubernetes, etc.) or, more popularly, through Graphical User Interfaces (GUIs) provided for these tools. Section 7.1 describes the available GUIs for monitoring deployments.

On top of these monitoring tools, data analytics tools can be applied to analyse the service logs and collect KPIs (Key Performance Indicators) that provide insight in the operation of the system and problematic parts, and provide hints to the resolution of issues. Such tools are discussed in Section 7.2.

Collection of feedback from the operation of the pilot sites is also important in order to understand if the system operates as expected, adhering to the system and user requirements. Feedback collection mechanisms are discussed in Section 7.3, while a more extensive analysis will be performed as part of T3.4 and reported in D3.7 – D3.9, “Technical Support Plan and Operations”.

7.1 Graphical User Interfaces (GUIs) for managing deployments

The tools for container composition and deployment management described in Section 6 provide the means to manage the deployments, e.g., starting and stopping services, viewing logs, etc. However, these functionalities are usually provided through the command line, making it hard for a user to have an overview. There are Graphical User Interfaces (GUIs) available that provide a comprehensive view of a deployment through a web interface, that greatly facilitate the deployment monitoring. Some of the most popular ones are discussed below.

7.1.1 Swarmpit

Swarmpit[76] is an open-source container management solution for monitoring and managing Docker swarm installations. It provides features such as service deployment, service management, service discovery, shared access across multiple users and integration with private Docker registries. A screenshot of Swarmpit can be seen in Figure 7.

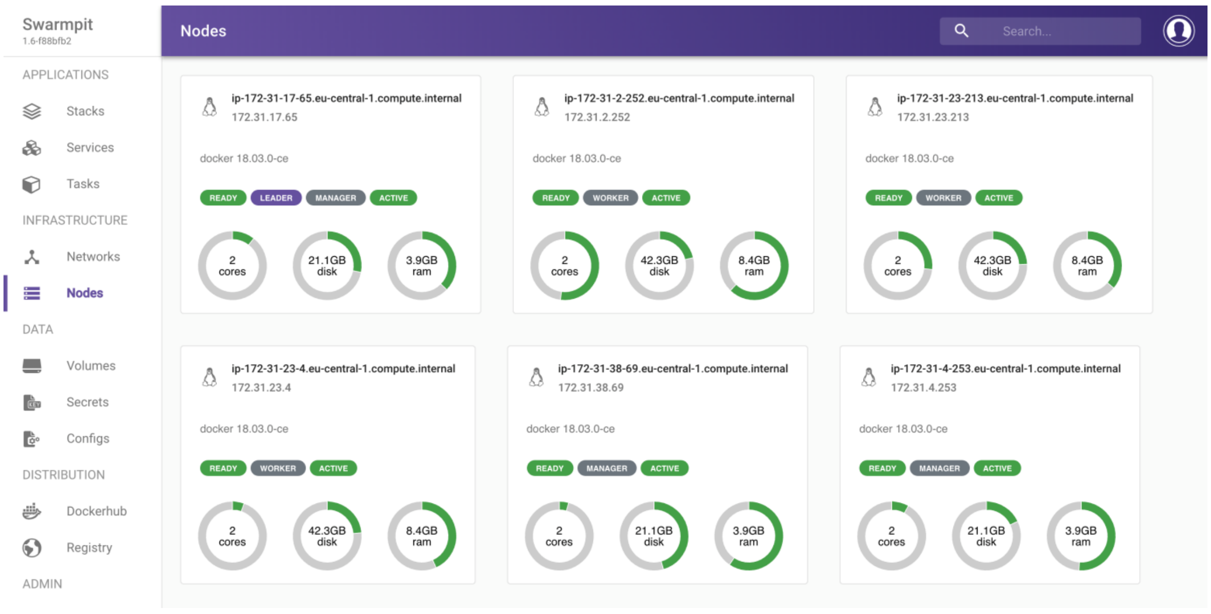

Swarmpit
1.6-f88bfb2
Nodes
Search...
APPLICATIONS
Stacks
Services
Tasks
INFRASTRUCTURE
Networks
Nodes
DATA
Volumes
Secrets
Configs
DISTRIBUTION
Dockerhub
Registry
ADMIN
ip-172-31-17-65.eu-central-1.compute.internal
172.31.17.65
docker 18.03.0-ce
READY LEADER MANAGER ACTIVE
2 cores 21.1GB disk 3.9GB ram
ip-172-31-2-252.eu-central-1.compute.internal
172.31.2.252
docker 18.03.0-ce
READY WORKER ACTIVE
2 cores 42.3GB disk 8.4GB ram
ip-172-31-23-213.eu-central-1.compute.internal
172.31.23.213
docker 18.03.0-ce
READY WORKER ACTIVE
2 cores 42.3GB disk 8.4GB ram
ip-172-31-23-4.eu-central-1.compute.internal
172.31.23.4
docker 18.03.0-ce
READY WORKER ACTIVE
2 cores 42.3GB disk 8.4GB ram
ip-172-31-38-69.eu-central-1.compute.internal
172.31.38.69
docker 18.03.0-ce
READY MANAGER ACTIVE
2 cores 21.1GB disk 3.9GB ram
ip-172-31-4-253.eu-central-1.compute.internal
172.31.4.253
docker 18.03.0-ce
READY MANAGER ACTIVE
2 cores 21.1GB disk 3.9GB ram

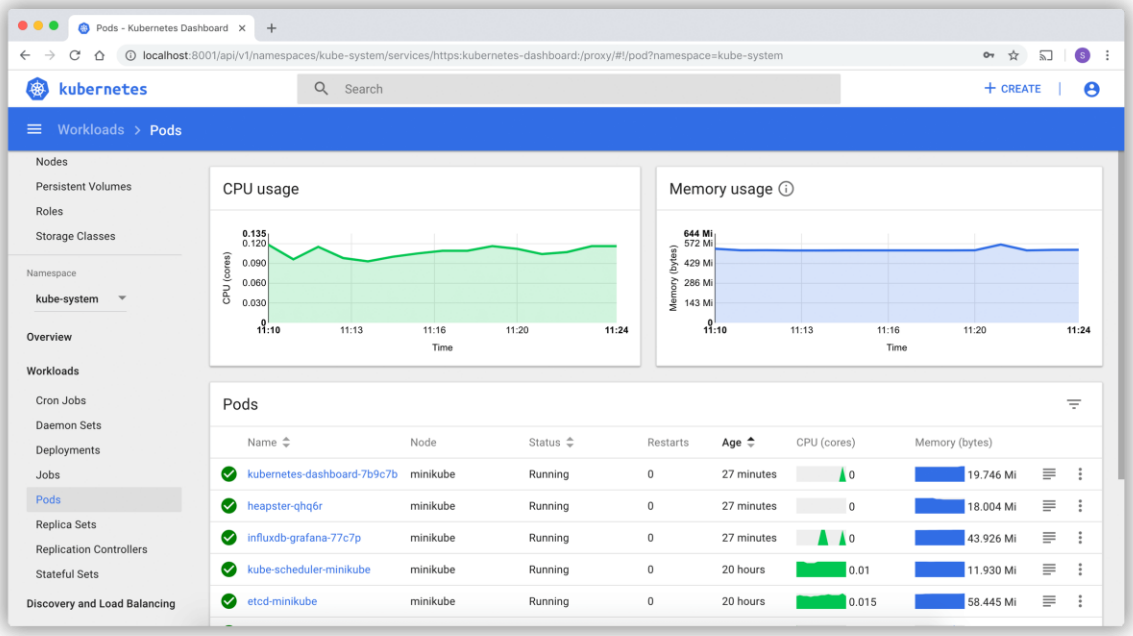
Figure 7: Screenshot of the Swarmpit GUI.

7.1.2 Kubernetes dashboard

Kubernetes dashboard[77] is the default dashboard of the Kubernetes orchestration framework. It can be used to deploy an application on a Kubernetes dashboard, monitor its operation, manage resources, scaling applications, starting pods, etc. A screenshot of the Kubernetes dashboard is shown in Figure 8.

[76] Swarmpit, https://swarmpit.io/ Last access June 2021.

[77] Kubernetes dashboard, https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/ Last access June 2021.



Pods

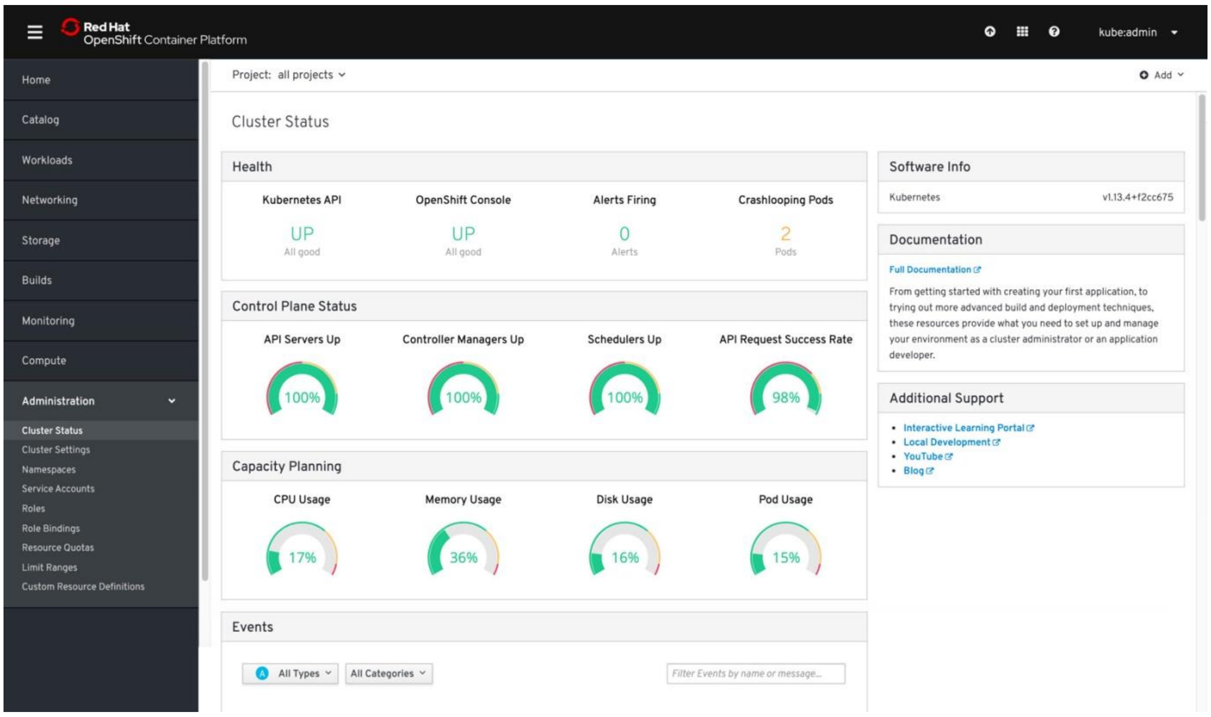
	Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
✓	kubernetes-dashboard-7b9c7b	minikube	Running	0	27 minutes	0	19.746 Mi
✓	heapster-qhq6r	minikube	Running	0	27 minutes	0	18.004 Mi
✓	influxdb-grafana-77c7p	minikube	Running	0	27 minutes	0	43.926 Mi
✓	kube-scheduler-minikube	minikube	Running	0	20 hours	0.01	11.930 Mi
✓	etcd-minikube	minikube	Running	0	20 hours	0.015	58.445 Mi

Figure 8: Screenshot of the Kubernetes dashboard GUI.

7.1.3 OpenShift

RedHat's OpenShift[78] is a container management GUI for the Kubernetes orchestrator. It offers several features, including automated installation and upgrades of the deployed applications, while focusing on security across the stack of deployed containers and through the lifecycle of an application. A screenshot of OpenShift can be seen in Figure 9.

[78] RedHat OpenShift, https://www.openshift.com/ Last access June 2021.



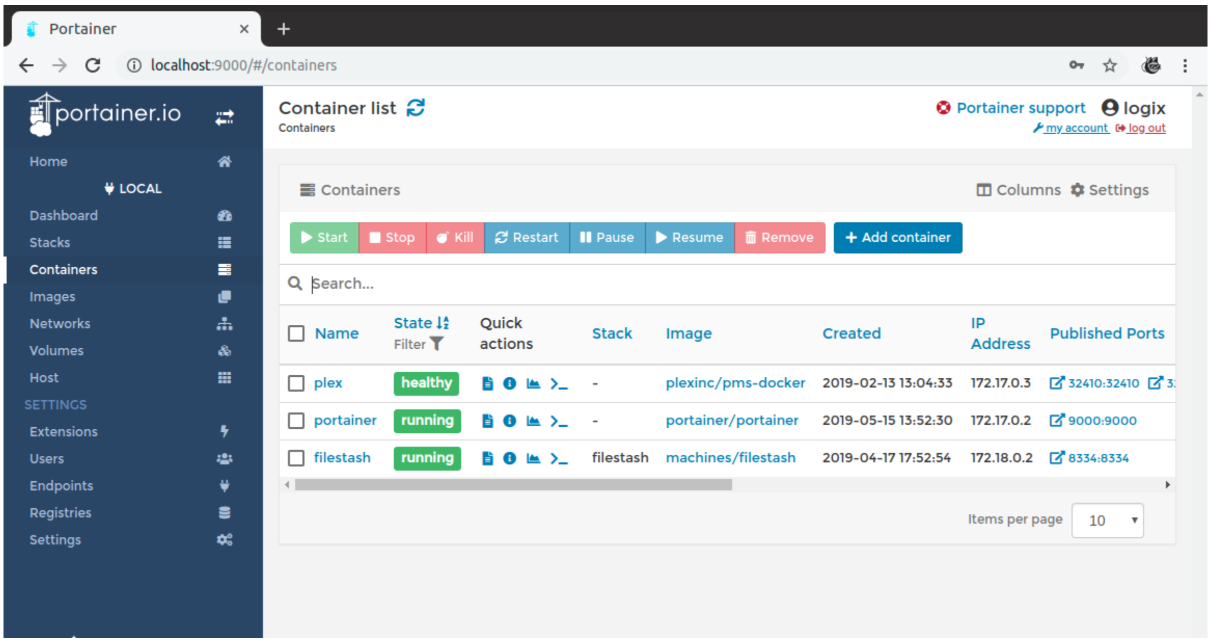
From getting started with creating your first application, to trying out more advanced build and deployment techniques, these resources provide what you need to set up and manage your environment as a cluster administrator or an application developer.

Figure 9: Screenshot of the OpenShift GUI.

7.1.4 Portainer

Portainer[79] is an open-source container management tool that can be used to monitor applications deployed using Docker, Docker swarm, Kubernetes, and other container orchestration frameworks. It can be used to deploy applications and view their status, providing secure access to authorized users. A screenshot of Portainer can be seen in Figure 10.

[79] Portainer, https://www.portainer.io/ Last access June 2021.



Name	State Filter	Quick actions	Stack	Image	Created	IP Address	Published Ports
plex	healthy		-	plexinc/pms-docker	2019-02-13 13:04:33	172.17.0.3	32410:32410 3
portainer	running		-	portainer/portainer	2019-05-15 13:52:30	172.17.0.2	9000:9000
filestash	running		filestash	machines/filestash	2019-04-17 17:52:54	172.18.0.2	8334:8334

Figure 10: Screenshot of the Portainer GUI.

7.1.5 Comparison

The characteristics of the deployment management GUIs presented in the previous sections are summarized and compared in Table 5. All GUIs offer a wealth of features for deployment monitoring and management, so the decision of the one to use is mostly based on free availability, the kinds of orchestrators supported and on the familiarity of ODIN's technical partners. Portainer is selected to be used in ODIN, since it can work with several types of orchestrators, including Docker swarm and Kubernetes, which will be used in ODIN.

Table 5: Comparison of deployment management GUIs.

	Swarmpit	Kubernetes dashboard	OpenShift	Portainer
Open-source	yes	yes	no	yes
Supported orchestrators	Docker swarm	Kubernetes	Kubernetes	Docker, Docker swarm, Kubernetes, etc.
Built-in features	high	high	high	high

7.2 Operation monitoring KPIs

Despite the capabilities of the system management tools (as described in Section 7.1), which already are capable of monitoring many container-wide and system-wide metrics, there will be the need to monitor some specific KPIs of the platform and services running on top which will not be covered by these tools. Task 4.6 will tackle this need from a resource perspective, trying to bridge technology and management monitoring.

From a functional perspective, monitoring all KPIs, including container-wide and system-wide, as well as service-specific and business metrics, is extremely important. It is the only way to determine if the system is running as it should, analysing any problems, auditing and making data-driven decisions at all levels. Additionally, the unification of these metrics into a single process will aid in the monitoring, reporting and analysis process, by providing perspective and context as well as extended services, such as anomaly detection, preventive alerts, corrective action proposal, and support requesting, among many other.

There are many existing open source system monitoring tools. There are extensible concepts such as using InfluxDB[80] for storing generic time series; or Prometheus[81] which also offers different access modes, queries for data and alerting. These tools offer integration with many other different sources and platforms such as the popular dashboard Grafana[82]. Other monitoring systems offer a more out-of-the-box experience, like Nagios[83] or Zabbix[84], however they are more system centric and less flexible.

Another important aspect of monitoring is log management. There are many open source centralization, parsing and processing systems, including the following:

- Elastic Stack[85], commonly abbreviated as ELK for Elasticsearch, Logstash, and Kibana
- Graylog[86]
- Fluentd[87]
- NXlog[88]

They are all extremely configurable and adaptable, particularly to containerized environments. Most offer additional features for searching within logs, linking different logs, providing anomaly detection, as well as parsing for further metric extraction.

It should be noted that the KPI collection tools described above are not limited to the DevOps pipelines. The same tools can and will be used for the ODIN platform as a whole. More detailed

[80] InfluxDB, https://www.influxdata.com/ Last access June 2021.

[81] Prometheus, https://prometheus.io/ Last access June 2021.

[82] Grafana, https://grafana.com/ Last access June 2021.

[83] Nagios, https://www.nagios.org/ Last access June 2021.

[84] Zabbix, https://www.zabbix.com/ Last access June 2021.

[85] Elastic Stack, https://www.elastic.co/ Last access June 2021.

[86] Graylog, https://www.graylog.org/ Last access June 2021.

[87] Fluentd, https://www.fluentd.org/ Last access June 2021.

[88] NXlog, https://nxlog.co/ Last access June 2021.

descriptions of the available options and selected tools will be provided in the deliverables of WP3 and WP7, regarding the ODIN platform and the KPI evolution, respectively.

7.3 Collecting feedback from pilot sites

An important part of the operation procedures is the collection of feedback in relation to the incidence management. All the procedures addressing the management of issues, bugs or requests will be fully specified in T3.4 and reported in the corresponding deliverables.

One of the goals of the task will be to work as closely as possible to the real-life operation, as such, we are proposing to take as basis the ITIL v4 framework, which describe an operative model for the delivery of technological services and products.

In its new version it reflects recent trends in software development and IT operations and includes advice on how to apply philosophies such as Agile, DevOps and Lean in the domain of service management. This new, more flexible version of ITIL has a more holistic approach to service management and focuses on "end-to-end service management, that is, from demand to value." Although it contains a total of 34 practices (14 general management practices, 17 service management practices and 3 technical management practices) in this case we will only apply Service management practices:

- Service desk: The purpose of this practice is to capture the demand for resolution of incidents and service requests. It should also be the entry point and single point of contact for the service provider with all its users.
- Incident management: The purpose of the incident management practice is to minimize the negative impact of incidents by restoring normal service operation as quickly as possible.
- Service request management: The purpose of the service request management practice is to support the agreed quality of a service by handling all predefined and user-initiated service requests in an effective and user-friendly manner.

There are multiple tools to implement the procedures that will be defined to provide support to the pilots' sites during the deployment and operation of the experiments. The final selection will be made in T3.4 according to the analysis of requirements that are being gathered from the different stakeholders. An initial identification and comparison of features of tools for service desk management is provided below.

7.3.1 Faveo Helpdesk

Faveo[89] is a free web-based ticketing system build on the Laravel framework, it provides businesses with an automated help desk system. It was released as open-source software under the OSL-3.0 license.

Its main features include the seamless email integration, notification management, email and in-app notification, integrated with multiple platforms and customizable.

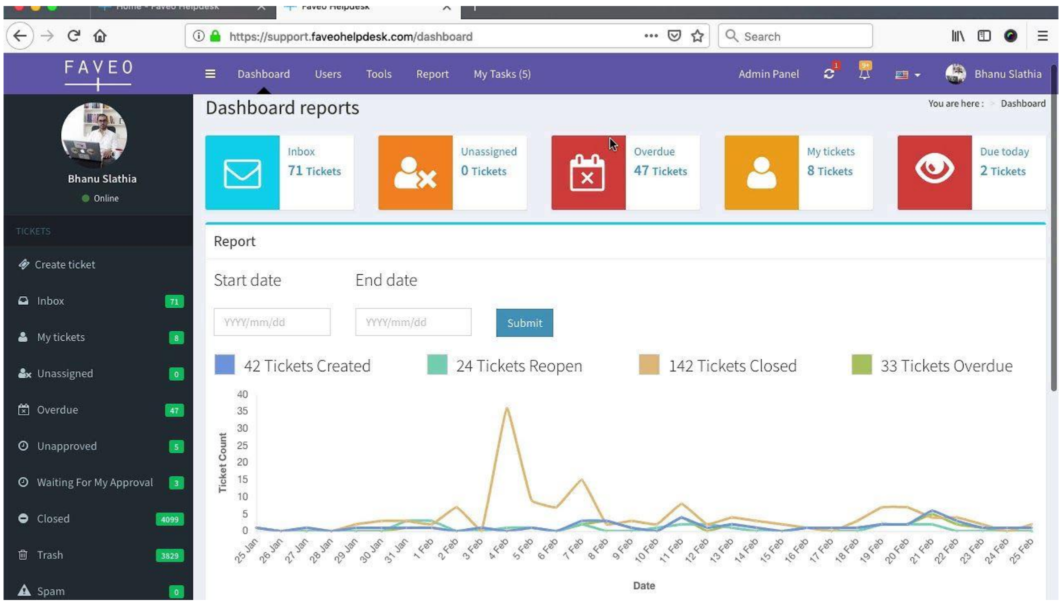


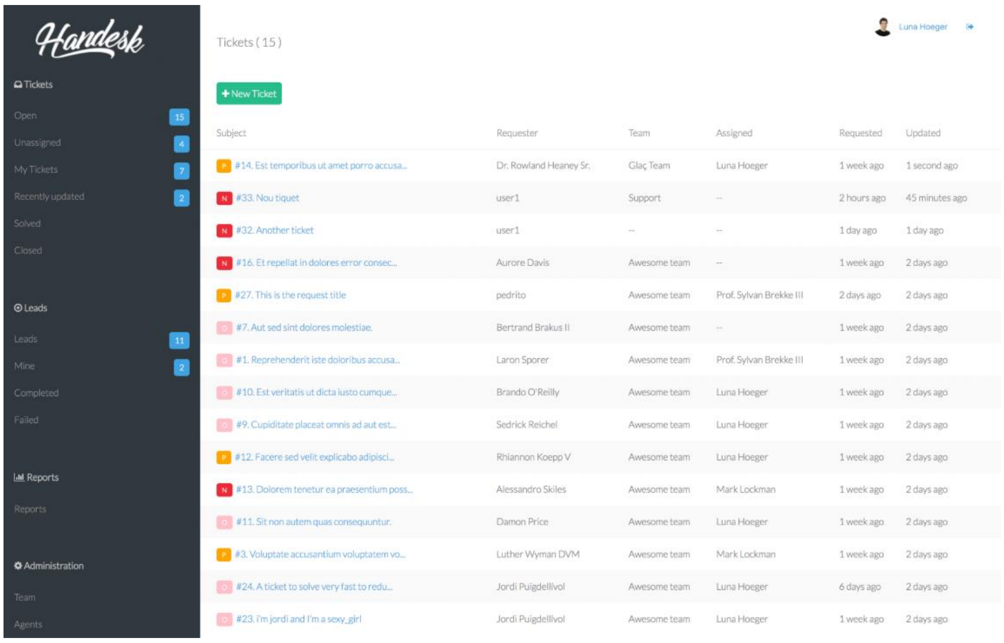
Figure 11: Screenshot of the Faveo Helpdesk GUI.

7.3.2 Handesk

Handesk[90] is a modular self-hosted powerful ticketing system, it provides with multiple teams, multiple users, easy and efficient reporting. It supports multi-language, email integration, in-app notificacion and reporting. It has been released as an open-source solution under MIT license.

[89] https://www.faveohelpdesk.com/ Last access June 2021

[90] http://handesk.io/ Last access June 2021



Tickets (15)

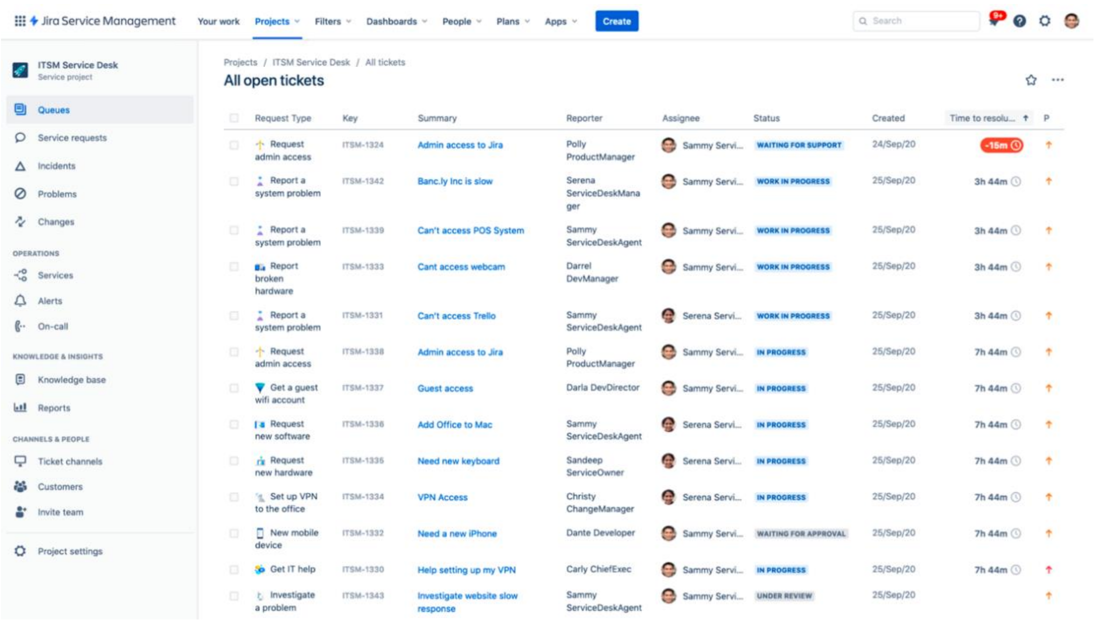
Subject	Requester	Team	Assigned	Requested	Updated
P #14. Est temporibus ut amet porro accusa...	Dr. Rowland Heaney Sr.	Glaç Team	Luna Hoeger	1 week ago	1 second ago
N #33. Nou tiquet	user1	Support	--	2 hours ago	45 minutes ago
N #32. Another ticket	user1	--	--	1 day ago	1 day ago
N #16. Et repellat in dolores error consec...	Aurore Davis	Awesome team	--	1 week ago	2 days ago
P #27. This is the request title	pedrito	Awesome team	Prof. Sylvan Brekke III	2 days ago	2 days ago
O #7. Aut sed sint dolores molestiae.	Bertrand Brakus II	Awesome team	--	1 week ago	2 days ago
O #1. Reprehenderit iste doloribus accusa...	Laron Sporer	Awesome team	Prof. Sylvan Brekke III	1 week ago	2 days ago
O #10. Est veritatis ut dicta iusto cumque...	Brando O'Reilly	Awesome team	Luna Hoeger	1 week ago	2 days ago
O #9. Cupiditate placeat omnis ad aut est...	Sedrick Reichel	Awesome team	Luna Hoeger	1 week ago	2 days ago
P #12. Facere sed velit explicabo adipisci...	Rhiannon Koepp V	Awesome team	Luna Hoeger	1 week ago	2 days ago
N #13. Dolorem tenetur ea praesentium poss...	Alessandro Skiles	Awesome team	Mark Lockman	1 week ago	2 days ago
O #11. Sit non autem quas consequuntur.	Damon Price	Awesome team	Luna Hoeger	1 week ago	2 days ago
P #3. Voluptate accusantium voluptatem vo...	Luther Wyman DVM	Awesome team	Mark Lockman	1 week ago	2 days ago
O #24. A ticket to solve very fast to redu...	Jordi Puigdellívol	Awesome team	Luna Hoeger	6 days ago	2 days ago
O #23. i'm jordi and I'm a sexy_girl	Jordi Puigdellívol	Awesome team	Luna Hoeger	1 week ago	2 days ago

Figure 12: Screenshot of the Handesk GUI.

7.3.3 Jira Service Desk

Jira Service Management[91] is a collaborative IT service management (ITSM) solution that enables to create multiple projects to track and handle customer support requests and incidents. It comprises features to provide request management, incident management, change management, asset management and knowledge management among others. It can be self-hosted or cloud based and has a free plan for up to 3 agents. It also allows multiple integrations through its REST API and open platform.

[91] https://www.atlassian.com/software/jira/service-management Last access June 2021



Projects / ITSM Service Desk / All tickets

All open tickets

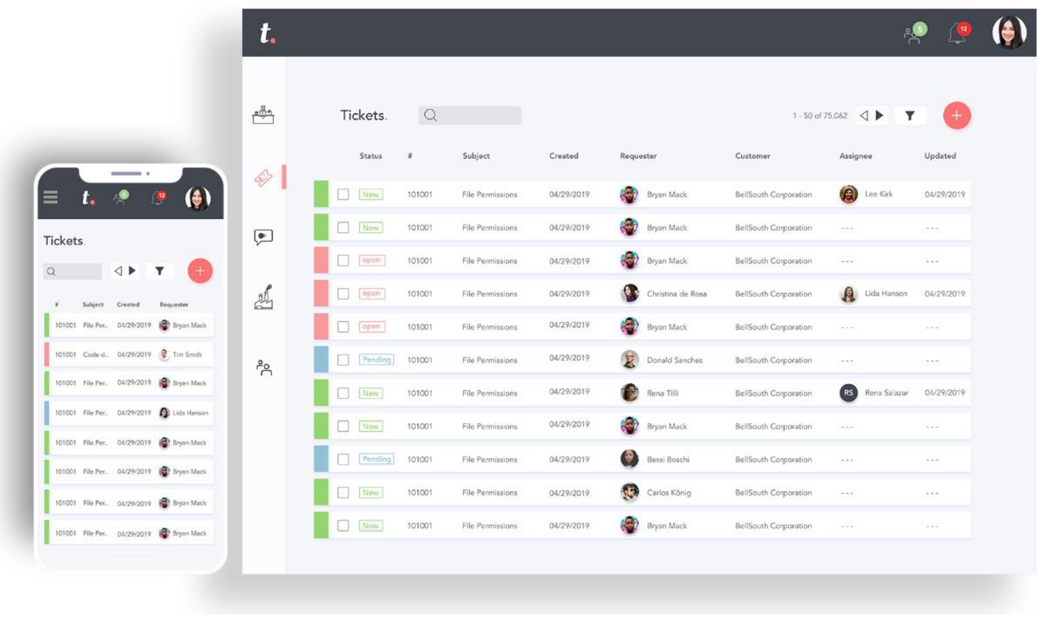
Request Type	Key	Summary	Reporter	Assignee	Status	Created	Time to resolu...	P
Request admin access	ITSM-1324	Admin access to Jira	Polly ProductManager	Sammy Servi...	WAITING FOR SUPPORT	24/Sep/20	-15m	↑
Report a system problem	ITSM-1342	Banc.ly Inc is slow	Serena ServiceDeskManager	Sammy Servi...	WORK IN PROGRESS	25/Sep/20	3h 44m	↑
Report a system problem	ITSM-1339	Can't access POS System	Sammy ServiceDeskAgent	Sammy Servi...	WORK IN PROGRESS	25/Sep/20	3h 44m	↑
Report broken hardware	ITSM-1333	Cant access webcam	Darrel DevManager	Sammy Servi...	WORK IN PROGRESS	25/Sep/20	3h 44m	↑
Report a system problem	ITSM-1331	Can't access Trello	Sammy ServiceDeskAgent	Serena Servi...	WORK IN PROGRESS	25/Sep/20	3h 44m	↑
Request admin access	ITSM-1338	Admin access to Jira	Polly ProductManager	Sammy Servi...	IN PROGRESS	25/Sep/20	7h 44m	↑
Get a guest wifi account	ITSM-1337	Guest access	Darla DevDirector	Sammy Servi...	IN PROGRESS	25/Sep/20	7h 44m	↑
Request new software	ITSM-1336	Add Office to Mac	Sammy ServiceDeskAgent	Serena Servi...	IN PROGRESS	25/Sep/20	7h 44m	↑
Request new hardware	ITSM-1335	Need new keyboard	Sandeep ServiceOwner	Serena Servi...	IN PROGRESS	25/Sep/20	7h 44m	↑
Set up VPN to the office	ITSM-1334	VPN Access	Christy ChangeManager	Serena Servi...	IN PROGRESS	25/Sep/20	7h 44m	↑
New mobile device	ITSM-1332	Need a new iPhone	Dante Developer	Sammy Servi...	WAITING FOR APPROVAL	25/Sep/20	7h 44m	↑
Get IT help	ITSM-1330	Help setting up my VPN	Carly ChiefExec	Sammy Servi...	IN PROGRESS	25/Sep/20	7h 44m	↑
Investigate a problem	ITSM-1343	Investigate website slow response	Sammy ServiceDeskAgent	Sammy Servi...	UNDER REVIEW	25/Sep/20		↑

Figure 13: Screenshot of the Jira Service Desk GUI.

7.3.4 Trudesk

Trudesk[92] is a complete self-hosted open-source solution for a help desk built with Node.JS and MongoDB. It has real-time tickets and updates, multiplatform design, live support chat, in-app notification and email integration. Trudesk is licensed under the Apache License, Version 2.0.

[92] https://trudesk.io/ Last access June 2021



Tickets.

#	Subject	Created	Requester
101001	File Per..	04/29/2019	Bryan Mack
101001	Code d...	04/29/2019	Tim Smith
101001	File Per..	04/29/2019	Bryan Mack
101001	File Per..	04/29/2019	Lida Hanson
101001	File Per..	04/29/2019	Bryan Mack
101001	File Per..	04/29/2019	Bryan Mack
101001	File Per..	04/29/2019	Bryan Mack
101001	File Per..	04/29/2019	Bryan Mack

Tickets.

1 - 50 of 75,062

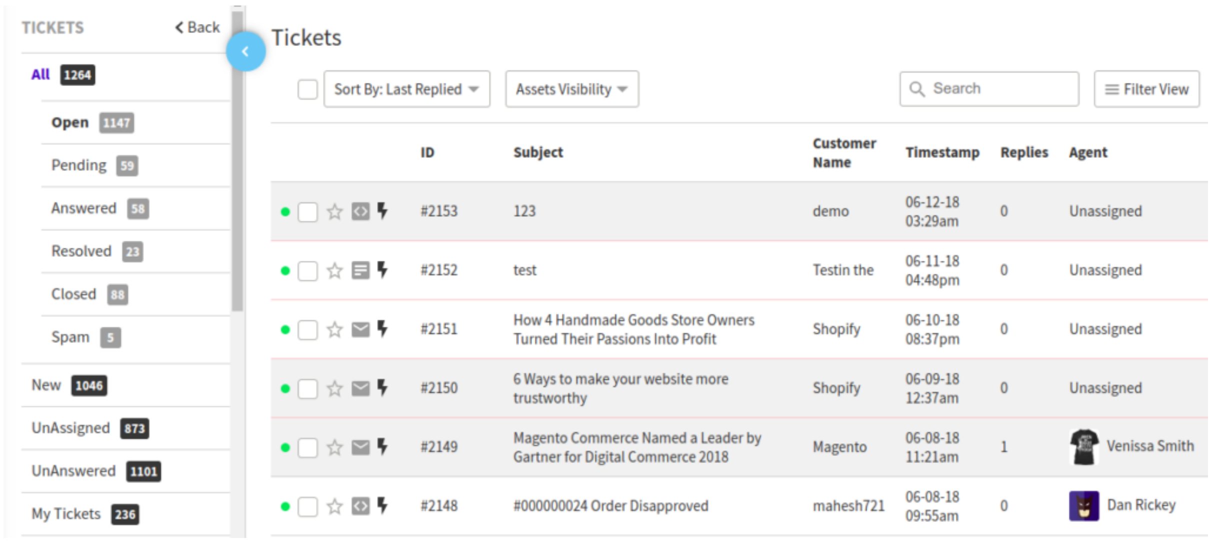
Status	#	Subject	Created	Requester	Customer	Assignee	Updated
New	101001	File Permissions	04/29/2019	Bryan Mack	BellSouth Corporation	Lee Kirk	04/29/2019
New	101001	File Permissions	04/29/2019	Bryan Mack	BellSouth Corporation	---	---
open	101001	File Permissions	04/29/2019	Bryan Mack	BellSouth Corporation	---	---
open	101001	File Permissions	04/29/2019	Christina de Rosa	BellSouth Corporation	Lida Hanson	04/29/2019
open	101001	File Permissions	04/29/2019	Bryan Mack	BellSouth Corporation	---	---
Pending	101001	File Permissions	04/29/2019	Donald Sanchez	BellSouth Corporation	---	---
New	101001	File Permissions	04/29/2019	Rena Tilli	BellSouth Corporation	RS Rena Salazar	04/29/2019
New	101001	File Permissions	04/29/2019	Bryan Mack	BellSouth Corporation	---	---
Pending	101001	File Permissions	04/29/2019	Bessi Boschi	BellSouth Corporation	---	---
New	101001	File Permissions	04/29/2019	Carlos König	BellSouth Corporation	---	---
New	101001	File Permissions	04/29/2019	Bryan Mack	BellSouth Corporation	---	---

Figure 14: Screenshot of the Trudesk GUI.

7.3.5 UVDesk

UVdesk[93] is a free open-source helpdesk ticket system, released as open-source software under MIT License. It is highly customizable and provides knowledge base, integration with email and workflow capabilities.

[93] https://www.uvdesk.com/en/opensource/ Last access June 2021



- All 1264
  - Open 1147
  - Pending 59
  - Answered 58
  - Resolved 23
  - Closed 88
  - Spam 5
- New 1046
- UnAssigned 873
- UnAnswered 1101
- My Tickets 236

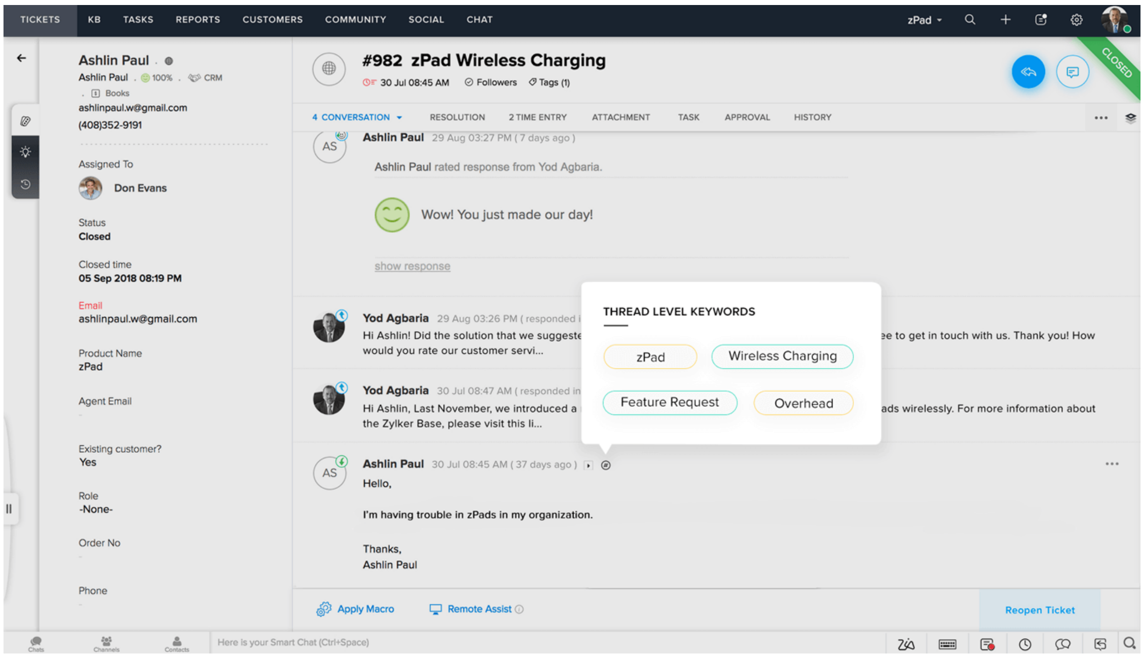
ID	Subject	Customer Name	Timestamp	Replies	Agent
#2153	123	demo	06-12-18 03:29am	0	Unassigned
#2152	test	Testin the	06-11-18 04:48pm	0	Unassigned
#2151	How 4 Handmade Goods Store Owners Turned Their Passions Into Profit	Shopify	06-10-18 08:37pm	0	Unassigned
#2150	6 Ways to make your website more trustworthy	Shopify	06-09-18 12:37am	0	Unassigned
#2149	Magento Commerce Named a Leader by Gartner for Digital Commerce 2018	Magento	06-08-18 11:21am	1	Venissa Smith
#2148	#000000024 Order Disapproved	mahesh721	06-08-18 09:55am	0	Dan Rickey

Figure 15: Screenshot of the UVDesk GUI.

7.3.6 Zoho Desk

Zoho Desk[94] is a cloud-based help desk software that allows to provide context-driven support. It provides comprehensive features and workflows for ticket management, assignment, categorization, prioritization, escalation, and more. It also features a knowledge base for ticket deflection through self-service, as well as easy-to-use dashboards to track quality metrics such as customer satisfaction and overall team performance.

[94] https://www.zoho.com/desk/ Last access June 2021



Ashlin Paul
Ashlin Paul . 100% . CRM
. Books
ashlinpaul.w@gmail.com
(408)352-9191

Assigned To
Don Evans

Status
Closed

Closed time
05 Sep 2018 08:19 PM

Email
ashlinpaul.w@gmail.com

Product Name
zPad

Agent Email
-

Existing customer?
Yes

Role
-None-

Order No
-

Phone
-

#982 zPad Wireless Charging
30 Jul 08:45 AM Followers Tags (1)

Ashlin Paul 29 Aug 03:27 PM (7 days ago)
Ashlin Paul rated response from Yod Agbaria.
Wow! You just made our day!
show response

Yod Agbaria 29 Aug 03:26 PM (responded i
Hi Ashlin! Did the solution that we suggeste … ee to get in touch with us. Thank you! How would you rate our customer servi...

Yod Agbaria 30 Jul 08:47 AM (responded in
Hi Ashlin, Last November, we introduced a … ads wirelessly. For more information about the Zylker Base, please visit this li...

Ashlin Paul 30 Jul 08:45 AM (37 days ago)
Hello,

I'm having trouble in zPads in my organization.

Thanks,
Ashlin Paul

Here is your Smart Chat (Ctrl+Space)

Figure 16: Screenshot of the Zoho Desk GUI.

7.3.7 Comparison

The characteristics of the tools for help desk presented in the previous sections are summarized and compared in Table 6. All tools offer appropriate features for supporting the collection of feedback from pilot sites, so the decision of the one to use will be mostly based on free availability, and the easiness to integrate with the rest of DevOps tools. Further analysis and the final choice will be done in T3.4 and reported in the corresponding deliverables.

Table 6: Comparison of Help Desk tools.

	Faveo Helpdesk	Handesk	Jira Service Desk	Trudesk	UVdesk	Zoho Desk
License	OSL 3.0	MIT	Proprietary	Apache 2.0	MIT	Proprietary
Ease of Use & set up	+++	+++	++	++++	+++	++
Built-in features	+++	++++	++++	++++	++++	++++
Integration	+++	+++	++++	+++	+++	++++
Hosting	Self-host / SaaS	Self-host / SaaS	SaaS	Self-host / SaaS	Self-host / SaaS	SaaS
Security	+++	+++	+++	+++	+++	++

7.4 ODIN guidelines

In ODIN, we will use **Portainer**, to manage the deployed services. Portainer will be available through the following URL:

https://portainer.odin-smarthospitals.eu

Portainer will be used to deploy applications, monitor their operation, view logs, start/stop services, etc. Access to the deployment managers of the pilot sites will be provided as soon as Portainer is installed in the pilot sites. Prior to that, Portainer will be available to test deployments deployed in ODIN's testing infrastructure (see Section 4.3).

Module developers should always document possible **monitoring metrics** of their modules. Whether these are front end reports, API endpoints, if they are actively reported, if they need further compilation from other sources (e.g. extraction from logs) or through any other mechanisms. Developers should also provide threshold information about these metrics, e.g. if metric X gets above/below Y then do Z; including failure hypothesis, possible corrective actions and tests.

Module developers should ensure all logs are reported to the standard output of the container, this way they can automatically be collected by the platform and centralized for further processing.

Feedback from the operation of the deployed components will be collected from the pilot sites using a **reporting and ticketing system** such as the ones presented in Section 7.3. The exact tool to use and the guidelines for feedback reporting will be decided through the activities of T3.4 and will be provided to the developers through deliverables D3.7-D3.9 "Technical Support Plan and Operations".

8 Pipeline orchestration

Sections 2 to 7 described the individual steps needed to establish a continuous workflow from the developer on one end to the end-user (pilot site) on the other end. The specific tools presented provide mechanisms to facilitate the fulfilment of each step. The final piece is to orchestrate the whole pipeline in a (semi-)automatic manner. This allows a change in the source code, e.g., a bug fix or a new functionality, to be automatically propagated all the way down to the end-user. This section describes the tools that will be used to automate the execution of the whole DevOps pipeline in ODIN.

8.1 Tools for CI/CD

This section describes some of the most used available tools for pipeline automation.

8.1.1 Jenkins

Jenkins[95] is an open-source automation server that is used to automatically build and deploy projects. Jenkins is used to define a pipeline of steps to be taken in order to build, test and deliver a software component, and can execute it automatically when a new version of the software is available in the source code versioning system used. In this way, it offers continuous delivery of software.

The definition of a pipeline is provided by the developer in a textual form, in a file named `Jenkinsfile`. The `Jenkinsfile` uses a user-friendly domain-specific language (DSL) to describe all the steps needed to perform the continuous delivery pipeline. The fact that the pipeline description is provided as a text file allows it to be committed to source code versioning tools along with the source code of the application, making it easy to maintain, update and use by developers.

The top-level concept of a `Jenkinsfile` is the `Pipeline`, which contains the description of a complete DevOps pipeline. A `Pipeline` consists of the following main parts:

- Node: A node is a machine on which the pipeline, or a part of it, will be executed. Jenkins supports several types of nodes, such as physical machines, virtual machines, Docker containers, Kubernetes nodes, etc.
- Stage: A stage is a conceptually distinct part of the pipeline, containing a set of steps to perform a particular sub-goal of the whole pipeline. A stage may represent e.g. the building phase, the testing phase, the deployment phase, etc.
- Step: A step is the basic element of a pipeline, representing a single task to perform. A step may e.g. run a shell command to build the source code, or call a testing framework

[95] Jenkins, https://www.jenkins.io/ Last access June 2021.

to test the built software. Jenkins provides a set of core types of steps, which is further extended with plugins to support a wide variety of available build/test/deployment frameworks.

An example Jenkinsfile can be seen below[96].

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'make'
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml'
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
```

The pipeline description contains two sections: the agent definition, which specifies which node the pipeline will run on (here any existing node), and the stages description. There are three stages defined:

- The “Build” stage. There is only one step in this stage which runs the “make” shell command to build the source code.
- The “Test” stage. There are two steps in this stage, one to run a shell command and another to call the JUnit testing framework, in order to test the built software.

[96] Example modified from https://www.jenkins.io/doc/book/pipeline/ Last access June 2021.

- The “Deploy” stage. There is one step in this stage to publish the built and tested software at an appropriate location.

There is a rich set of configuration options and step plugins at the disposal of the developer, allowing them to define a wide variety of workflows. The reader may find online relevant reference for the Jenkinsfile syntax and the available options and steps[97,98].

8.1.2 CircleCI

CircleCI[99] is a popular CI/CD tool that facilitates automation of a complete CI/CD pipeline. Similar to Jenkins, the definition of a workflow is specified in a textual file, commonly named `config.yml`, which can be subject to version control along with the source code.

CircleCI is directly integrated with Docker to provide isolated execution environments for the CI/CD steps. In contrast to Jenkins, in which the functionality of the steps is provided by plugins, CircleCI provides building and testing functionalities as part of its core, resulting in a more unified environment. Moreover, CircleCI provides the so called “orbs”, which are reusable and sharable packages of configuration options for common steps and projects, e.g. for installing a Node.js server or pushing images to cloud services. CircleCI also directly integrates with Bitbucket, GitHub, and GitHub Enterprise. To use the full functionalities of CircleCI, users need to pay a corresponding fee.

CircleCI uses a YAML format for the description of the CI/CD workflows. An example `config.yml` file is the following[100].

[97] Jenkins pipeline syntax, https://www.jenkins.io/doc/book/pipeline/syntax/ Last access June 2021.

[98] Jenkins pipeline steps, https://www.jenkins.io/doc/pipeline/steps/ Last access June 2021.

[99] CircleCI, https://circleci.com/ Last access June 2021.

[100] Example modified from https://circleci.com/docs/2.0/sample-config/ Last access June 2021.

```yaml
version: 2.1

# Define the jobs we want to run for this project
jobs:
  build:
    docker:
      - image: circleci/<language>:<version TAG>
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD
    steps:
      - checkout
      - run: echo "this is the build job"
  test:
    docker:
      - image: circleci/<language>:<version TAG>
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD
    steps:
      - checkout
      - run: echo "this is the test job"

# Orchestrate our job run sequence
workflows:
  build_and_test:
    jobs:
      - build
      - test
```

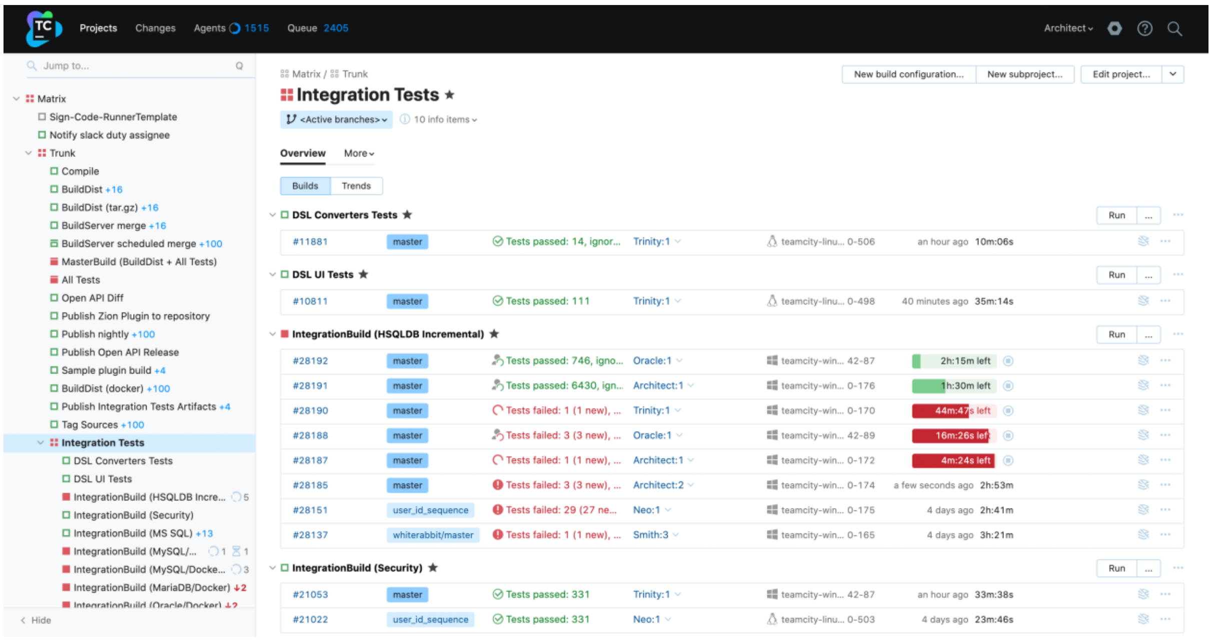
The file first defines the types of jobs that will be run in the pipeline, here the “build” and the “test” jobs, and how each is executed, describing the Docker containers and the steps to run. Then, the file defines the workflow, i.e. the order in which the jobs should be performed, here first the “build” job and then the “test” job.

8.1.3 TeamCity

TeamCity[101] is a comprehensive solution for CI/CD that allows the specification and management of a CI/CD pipeline through a graphical interface. It integrates with popular building tools such as Maven, NPM and Gradle (see Section 3.1) and facilitates the

[101] TeamCity, https://www.jetbrains.com/teamcity/ Last access June 2021.

management of build and testing steps through the GUI. It provides analysis of failures and visualizations of pipelines that make it easier for the developer to specify and monitor the pipeline. In addition to the visual interface, TeamCity allows the specification of the CI/CD pipeline as a script using the Kotlin language. A screenshot of the TeamCity tool is shown in Figure 17.



DSL Converters Tests ★

#11881	master	Tests passed: 14, ignor...	Trinity:1	teamcity-linu... 0-506	an hour ago	10m:06s

DSL UI Tests ★

#10811	master	Tests passed: 111	Trinity:1	teamcity-linu... 0-498	40 minutes ago	35m:14s

IntegrationBuild (HSQLDB Incremental) ★

#28192	master	Tests passed: 746, igno...	Oracle:1	teamcity-win... 42-87	2h:15m left	
#28191	master	Tests passed: 6430, ign...	Architect:1	teamcity-win... 0-176	1h:30m left	
#28190	master	Tests failed: 1 (1 new), ...	Trinity:1	teamcity-win... 0-170	44m:47s left	
#28188	master	Tests failed: 3 (3 new), ...	Oracle:1	teamcity-win... 42-89	16m:26s left	
#28187	master	Tests failed: 1 (1 new), ...	Architect:1	teamcity-win... 0-172	4m:24s left	
#28185	master	Tests failed: 3 (3 new), ...	Architect:2	teamcity-win... 0-174	a few seconds ago	2h:53m
#28151	user_id_sequence	Tests failed: 29 (27 ne...	Neo:1	teamcity-win... 0-175	4 days ago	2h:41m
#28137	whiterabbit/master	Tests failed: 1 (1 new), ...	Smith:3	teamcity-win... 0-165	4 days ago	3h:21m

IntegrationBuild (Security) ★

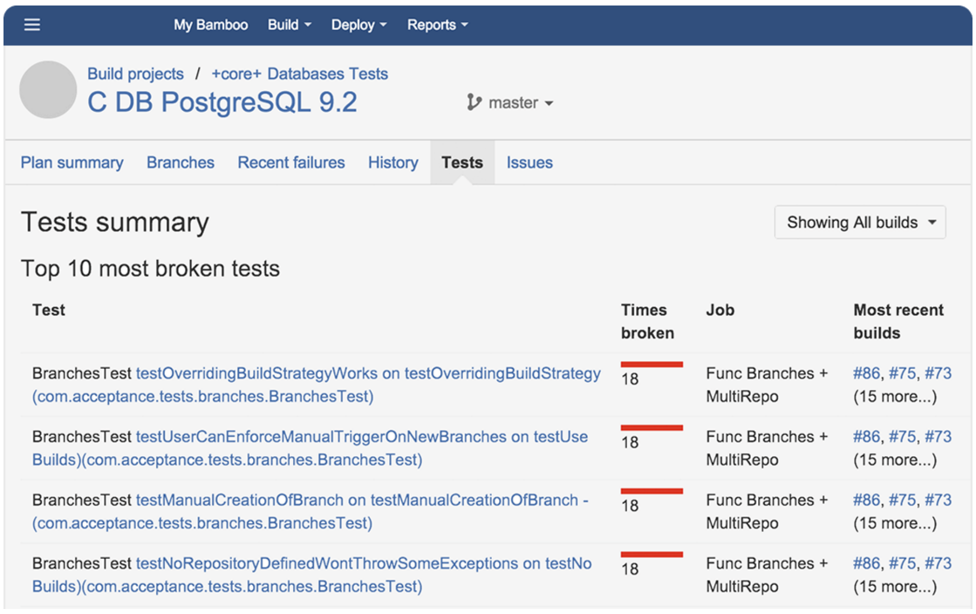
#21053	master	Tests passed: 331	Trinity:1	teamcity-win... 42-87	an hour ago	33m:38s
#21022	user_id_sequence	Tests passed: 331	Neo:1	teamcity-linu... 0-503	4 days ago	23m:46s

Figure 17: Screenshot of the TeamCity CI/CD tool.

8.1.4 Bamboo

Bamboo[102] is a CI/CD GUI for constructing build and testing pipelines. It supports multi-stage build plans, which can be executed upon source code commit, through the setup of appropriate triggers, as well as automated test runs. A screenshot of Bamboo is shown in Figure 18.

[102] Bamboo, https://www.atlassian.com/software/bamboo Last access June 2021.



Tests summary

Top 10 most broken tests

Test	Times broken	Job	Most recent builds
BranchesTest testOverridingBuildStrategyWorks on testOverridingBuildStrategy (com.acceptance.tests.branches.BranchesTest)	18	Func Branches + MultiRepo	#86, #75, #73 (15 more...)
BranchesTest testUserCanEnforceManualTriggerOnNewBranches on testUse Builds)(com.acceptance.tests.branches.BranchesTest)	18	Func Branches + MultiRepo	#86, #75, #73 (15 more...)
BranchesTest testManualCreationOfBranch on testManualCreationOfBranch - (com.acceptance.tests.branches.BranchesTest)	18	Func Branches + MultiRepo	#86, #75, #73 (15 more...)
BranchesTest testNoRepositoryDefinedWontThrowSomeExceptions on testNo Builds)(com.acceptance.tests.branches.BranchesTest)	18	Func Branches + MultiRepo	#86, #75, #73 (15 more...)

Figure 18: Screenshot of the Bamboo CI/CD tool.

8.1.5 GitLab

GitLab, the Git-based source code versioning tool, provides its own CI/CD mechanisms[103] that can be employed by developers to setup pipelines to run upon source code commit and push. GitLab pipelines are written in a YAML and can support multiple types of pipelines, such as directed acyclic graphs and parent-child pipelines. Pipelines support building, testing and production stages. GitLab also supports Auto DevOps[104], which automatically creates CI/CD pipelines by analysing the source code of the repository and creating appropriate build rules. An example GitLab YAML file can be seen below.

[103] Gitlab CI/CD, https://docs.gitlab.com/ee/ci/ Last access June 2021.

[104] GitLab Auto DevOps, https://docs.gitlab.com/ee/topics/autodevops/index.html Last access June 2021.

```
stages:
  - build
  - test
  - deploy

image: alpine

build_a:
  stage: build
  script:
    - echo "This job builds something."

test_a:
  stage: test
  needs: [build_a]
  script:
    - echo "This job tests something."

deploy_a:
  stage: deploy
  needs: [test_a]
  script:
    - echo "This job deploys something."
```

8.1.6 Comparison

The characteristics of the CI/CD pipeline orchestration tools presented in the previous sections are summarized and compared in Table 7. In ODIN, Jenkins has been selected as the CI/CD tool. Although the built-in set of its features is limited, it provides extensive functionalities through a wealth of available plugins. Furthermore, it allows the specification of the pipeline as a script that can be submitted to version control. Jenkins is supported by a wide community and is quite familiar to the technical members of the consortium, which are also reasons for its selection.

Table 7: Comparison of CI/CD pipeline orchestration tools.

	Jenkins	CircleCi	TeamCity	Bamboo	GitLab
Open source	Yes	No	No	No	No
Ease of Use & set up	Medium	Medium	Medium	Medium	Medium
Built-in features	3/5	4/5	4/5	4/5	4/5
Integration	Very Good	Medium	Good	Medium	Good
Hosting	On premise & Cloud	On premise & Cloud	On premise	On premise & Bitbucker as Cloud	On premise & Cloud
Free Version	Yes	Yes	Yes	Yes	Yes

8.2 ODIN guidelines

In ODIN, we will use **Jenkins** for CI/CD automation. A Jenkins pipeline will ideally consist of the following stages:

- Building, using build automation tools such as the ones described in Section 3.1;
- Testing, specifying both unit and integration tests, where applicable, as described in Section 4;
- Containerization, running the appropriate Docker commands to build component images, according the provided Dockerfiles, as described in Section 3.2;
- Releasing components to ODIN's Docker registry, as described in Section 5.4;
- Deploying applications to the pilot sites, according to the specification of the available docker-compose.yml files, as described in Section 6.3;

The Jenkins pipeline will be triggered upon pushing a new version of the source code of a component to ODIN's source code GitLab repositories. In case of pipeline failure, the developers and the DevOps team will be notified in order to proceed to the appropriate actions.

The specification of the CI/CD pipeline for a particular application of ODIN will be written in a Jenkinsfile, which will be submitted to ODIN's **GitLab** for version control. The writing and management of the Jenkinsfiles of all ODIN services will be under the responsibility of the DevOps team. The DevOps team, in coordination with component developers and pilot site deployment administrators, will compose the Jenkinsfile pipelines and submit them to ODIN's GitLab. A separate repository will be devoted for the Jenkinsfiles, which will be managed by the DevOps team.

The available Jenkins files will also be managed through the Jenkins server. Access to this server will be provided to the DevOps team, as well as to development teams, as needed.

9 Horizontal services

This section covers services that span the whole DevOps infrastructure horizontally. These include the security infrastructure for DevOps, component documentation, the DevOps home page providing access to all services, and the approaches to ensure high quality throughout the DevOps pipeline.

9.1 Security mechanisms for DevOps

This section describes the security mechanisms used to ensure that the DevOps infrastructure of ODIN is used only by authorized users within the consortium. The DevOps infrastructure is different from the ODIN Platform (which will be better analysed in much more depth in D3.4), being restricted to project members and community that will be working on the development of the different aspects of the ODIN technology.

As such, the security mechanisms need only to protect the development process and its results. This means ensuring only trusted entities can view (download, read), update (modify, write) or execute the following assets produced by ODIN:

- Source code, especially critical code-base
- Binaries, including libraries and images
- Pipeline configuration, execution, logs, and results
- Documentation, particularly sensitive specifications and reports
- Support and tickets
- Testing infrastructure
- DevOps infrastructures and secrets (e.g. access keys to other services)

The access to these assets must be restricted, at least during the run of the project. As such, each of these assets must be configured with access control mechanisms, only authorising trusted entities to them. These entities must be authenticated so they can claim their trusted status.

All these assets also need to be verifiable, i.e. trusted entities should be able to trust that the integrity of these assets has not been compromised by possible third parties, in an attempt to disrupt or gain access to the development process.

Finally access to all these assets must be confidential. All communications between the trusted entities and the asset repositories or infrastructure must be encrypted, so as not to disclose critical aspects of the development or its process. Of course, once the trusted entity has access to these assets, they are trusted not to share, inadvertently or otherwise, these assets with other non-trusted (or even trusted) entities; and inform if they do.

9.1.1 Single Sign On and Authorisation service: Keycloak

Keycloak[105] is an open source tool which manages cloud authentication and authorisation. Centralized authentication means that all trusted entities can be registered in a single point, offered as Single Sign-On (SSO), to access the different services, without the risk of confusing entities or human error. Keycloak is also capable of loading its user base from different systems, effectively federating user identity, even though common identity providers such as Google, Facebook, GitHub, or Twitter.

All DevOps infrastructure services can connect with Keycloak authentication using OpenID connect, SAML 2.0 or OAuth 2.0; all very common authentication and authorisation mechanisms. Keycloak manages roles for all users, so that the services can enforce these roles, but Keycloak can go beyond role-based access control and also implement complex access control policies.

9.1.2 Public Key Infrastructure: SKS keyserver & Docker notary server

Anyone can create a private-public key pair, however trust must be built and the public key disseminated in order for the encryption to be effective, or signature to be validated. To aid in this, a Public Key Infrastructure (PKI) needs to be implemented, which will take care of distributing public keys as well as maintaining the trust chain in these keys. OpenPGP (RFC 4880) is a de facto standard for email encryption and signing; this is why it is also used for digital signing of Git commits[106] as well as for signing packages for building automation tools. For OpenPGP, the standard PKI is built upon a web of servers known as *keyservers*, which are queried for exchanging keys. The most common keyserver is SKS[107], but there are alternatives such as Skier[108]. In both cases, a web interface is used to upload public keys, which can be restricted to trusted entities (using access control provided by Keycloak).

As with any type of content, Docker images may be transmitted securely through Transport Later Security (TLS), however this does not guarantee that the content itself has not been tampered with. Docker Content Trust uses a public-private key schema to allow the signing and verification of Docker images[109]. However, the last update (as of June 2021) of Docker Content

[105] Keycloak, https://www.keycloak.org/ Last access June 2021.

[106] Git signing tools, https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work Last access June 2021.

[107] SKS Keyserver, https://github.com/SKS-Keyserver/sks-keyserver Last access June 2021.

[108] Skier, https://github.com/SkierPGP/Skier Last access June 2021.

[109] Docker trust, https://docs.docker.com/engine/security/trust/ Last access June 2021.

Trust was the 10th of April of 2018, rendering it insecure. There are two alternatives: Skopeo[110] which could be used to sign Docker images with OpenPGP keys, and CodeNotary[111] which has its own system for signing Docker images. At the moment none of the two are definitive.

9.1.3 Transport Layer Security: X509 certification

All DevOps infrastructure services are accessed through HTTP, which can be protected through Transport Later Security (TLS) or commonly known as HTTPS. In order to achieve so, x509 certificates have to be installed and configured so that the HTTPS server presents them to the client browser. However, the client will not trust this connection unless the certificate is signed by any of a trusted set of Certificate Authorities (CA), or unless a chain of trust (other CA whose certificate are signed by CA) with these trusted CA can be verified. Thus, it is imperative that the installed certificate is verifiable this way.

Fortunately initiatives such as Let's encrypt[112], provide a free and secure service through the use of the ACME protocol. Most of the tools used for DevOps already implement the ACME protocol to obtain a valid x509 certificate. If not, the Certbot tool[113] can be used (and automated) to obtain valid certificates.

9.1.4 ODIN guidelines

A **Keycloak** instance will be deployed in the DevOps infrastructure, allowing users to manage their credentials, and other tools to interoperate with it in the following URL:

https://account.odin-smarthospitals.eu

A SKS keyserver will be deployed, allowing only valid Keycloak users to upload OpenPGP keys (query and download of keys will be public) in the following URL:

https://sks.odin-smarthospitals.eu

The ODIN DevOps security infrastructure will include at least the user roles of Table 8. An individual user may have more than one role. Depending on the user role, the DevOps home page (see Section 9.3) will be adapted to offer links to the accessible servers.

[110] Skopeo, https://github.com/containers/skopeo Last access June 2021.

[111] CodeNotary, https://www.codenotary.com/ Last access June 2021.

[112] Letsencrypt, https://letsencrypt.org/ Last access June 2021.

[113] Certbot, https://certbot.eff.org/ Last access June 2021.

Table 8: ODIN DevOps user roles.

Role	Accessible DevOps infrastructure
DevOps manager	All
Developer	Gitlab, Docker registry
Tester	Docker registry, Portainer, Docker swarm, Kubernetes, testing infrastructure, operation monitoring, ticketing
Deployer	Portainer, Docker swarm, Kubernetes, operation monitoring, ticketing
Pipeline manager	Jenkins
Security administrator	Keycloak, SKS
Third-party (e.g. open caller)	GitLab, Docker registry, Portainer, Docker swarm, Kubernetes, operation monitoring, ticketing

All DevOps services (see 0) will be encrypted using valid x509 certificates. Trusted entities (developers, pipelines, partners, etc.) will be encouraged to follow common sense security policies:

- Credentials, certificates, and keypairs are personal and untransferable.
- Sensitive information should never be shared with third parties.
- Sensitive information should never be published (in online comments, commits, pipelines, Docker images nor any other kind of content unless it is properly secured).

A breach of these conditions should immediately be reported to ODIN DevOps manager via a ticketing system in order to properly mitigate the security risks.

In order to maintain integrity, all Git tags should be signed by any member registered in the SKS keyserver. The signature will cover the tag content as well as all other previous commits. Signatories are encouraged to verify and validate all, especially the security considerations.

Integrity should also be maintained for binaries and Docker images. The Integration Manager will verify that all stable releases, including Git tags, Docker images and any other persistent content, are appropriately signed before releasing.

9.2 Documentation

Documentation of the developed components, systems and their interactions is important to facilitate the use of components by other developers or by the same developer at later stages. Component documentation runs in parallel to component development and deployment, documenting the functionality of the components, and providing instructions for their installation and use.

In ODIN, a knowledge base will be created to hold all documentation regarding the developed components, in the form of a wiki that will be constantly be updated during the course of the project as new components are made available or existing ones are updated. Part of this documentation will be created as part of the source code itself, making use of per-language documentation frameworks (see related guidelines in Section 2.4). The implementation of the ODIN knowledge base and of the full set of guidelines regarding component documentation is part of the activities of T3.4 and will be reported in more details in deliverables D3.7 – D3.9 “Technical Support Plan and Operations”. In this section, these aspects are briefly described for completeness.

9.2.1 ODIN Knowledge Base

Knowledge management is an important aspect applicable to all stages of the DevOps cycle. According to ITIL 4 *“knowledge management aims to ensure that stakeholders get the right information, in the proper format, at the right level, and at the correct time, according to their access level and other relevant policies. This requires a procedure for the acquisition of knowledge, including the development, capturing, and harvesting of unstructured knowledge, whether it is formal and documented or informal and tacit knowledge”*[114]

This will be relevant to all type of stakeholders, from developers, to integrators, deployers and pilot site members, as well as end-users, and in order to be successful, it needs to be connected to their workflows, to enable a rich information architecture, and create consumable documentation for all stakeholders.

Knowledge is one of the project most valuable assets and open knowledge sharing will help all stakeholders to collaborate, create value and foster innovation around ODIN goals. In practical terms, ODIN will define and collect all needed data and information and aggregate it in a single self-serve online library wiki, available to all stakeholders through the DevOps landing page.

The information architecture and type of content for the knowledge base will be further define in T3.4 and reported in the corresponding deliverables.

9.2.2 Component documentation

Documentation issues are as vital to the success of any project as the code itself[115]. In particular, source code documentation has always been a topic that generated much debate and opposing opinions. However, today there is a very strong current of opinion stating that it is more important that developers follow good practices, coding standards, principles such as "Clean Code" (use self-describing function / class names, that variables have names that are readable and that we don't have to think too much to know what is happening), in short, that they program in such a way that their code does not need to be commented. Sometimes the necessary documentation is minimum and sometimes the unit tests themselves give them enough information for another developer to understand the code snippet.

Within T3.4, a common strategy for documentation will be established in collaboration with the developers’ teams to reach an agreement about the purpose of documenting code (to make the code readable by other programmers, to make the code usable, etc.). The goal will be to

[114] ITIL 4 Foundation, 5.1.4, Knowledge management. Axelos. https://www.axelos.com/store/book/itil-foundation-itil-4-edition.

[115] https://www.itpro.co.uk/606693/the-need-to-know-documentation-in-linux

establish a series of basic principles and then define the processes and finally the tools to use, as different components may require different types of documentation (i.e. API documentation, AI service documentation, etc.).

Although manual documentation will always be needed, there exist quite a number of tools that enable automatizing part of the process. Tools like Swashbuckle[116] or NSwag[117] can generate automatically Swagger documents based on the existing code. DocFx[118] can generate API and/or Markdown files based documentation, as well as GhostDoc[119] can generate and validate XML comments and create help files automatically.

9.3 DevOps home page

In order for the DevOps manager and the developers/deployers to handle all DevOps services, a centralized DevOps home page will be used as a “landing page” providing access to all services. The purpose of this home page is to provide access to the different types of services and to manage user authentication.

Landing pages can be implemented in an ad-hoc manner, as a separate component, providing a higher layer above several other GUIs. However, there are existing tools that can provide such functionality out-of-the-box. Organizr[120] is such a tool. It can be used to organize multiple services within the same screen, e.g., by putting them in different tabs or side-by-side. Moreover, it can be used to provide user access to specific tabs, a feature that can be used to design custom landing pages according to the type of user that logs in, providing different sets of functionalities.

The ODIN DevOps homepage will be available at the following URL:

https://dev.odin-smarthospitals.eu

It will provide access to the following DevOps services:

- GitLab, for source code management, along with separate parts for deployment configurations and Jenkins pipelines
- Docker registry, for viewing and managing ODIN’s Docker registry
- Testing infrastructure, for testing the developed components

[116] https://www.c-sharpcorner.com/article/swashbuckle-and-asp-net-core/

[117] https://github.com/RicoSuter/NSwag

[118] https://dotnet.github.io/docfx/

[119] https://submain.com/ghostdoc/

[120] Organizr, https://organizr.app/ Last access June 2021.

- Portainer, with corresponding Docker swarm / Kubernetes servers, for deployment monitoring
- KPI collection tool, for viewing and analysing monitored KPIs
- Feedback collection and ticketing tools, for collecting feedback from the pilot sites in the form of bug reports and other documents.
- Jenkins server, for DevOps pipeline construction and management
- Keycloak server, for security management
- ODIN knowledge base, for component and service documentation

Depending on the role of the authorized user (see Table 8), different subsets of the above services will be available through the DevOps homepage. The exact URLs of the landing page and all the sub-GUIs will be disseminated to the project partners once the corresponding tools are up and running.

Apart from providing access to the above listed services, the DevOps home page will also provide guidelines for the use of these services by developers, deployers and managers. For each type of service, it will provide links to the corresponding guidelines, as reported in this deliverable, which will be available through the project's Wiki.

A detailed list of all DevOps services that will be available through the DevOps homepage, with links to the associated guidelines, can be found in 10.

9.4 DevOps quality assurance

DevOps is designed for a continuous monitoring of development, testing and deployment activities. If these activities are carried out following best practices in standardization and well known Git patterns, a high product quality can be assured.

In the case of standards, the ISO 9000 family[121] defined a set of international standards on quality management and quality assurance. They are not specific to any one industry and can be applied to organizations of any size. Within ISO 9000, the ISO 9001[122] standard sets out the criteria for a quality management system and is the only standard in the family that can be certified to (although this is not a requirement). It can be used by any organization, large or small, regardless of its field of activity. In fact, there are over one million companies and organizations in over 170 countries certified to ISO 9001.

[121] ISO 9000 family standard, https://www.iso.org/iso-9001-quality-management.html Last access June 2021.

[122] ISO 9001, https://www.iso.org/standard/62085.html Last access June 2021.

The structure of ISO 9001 is divided into ten sections. The first three are introductory, while the last seven contain the requirements relating to the Quality Management System. Below is a summary of the seven main sections:

- Section 4: Context of the organization - This section talks about the requirements for understanding the organization in order to implement a Quality Management System (QMS). It includes the requirements for identifying internal and external problems, identifying stakeholders and their expectations, defining the purpose of the QMS and identifying the processes and how they interact.
- Section 5: Leadership - The leadership requirements concern the need for top management to be instrumental in the implementation of the QMS. Top Management must demonstrate commitment to the QMS by ensuring customer attention, defining and communicating the quality policy and assigning roles and responsibilities within the organization.
- Section 6: Planning - Top Management must also plan the ongoing operation of the QMS. It is necessary to evaluate the risks and opportunities of the QMS within the organization and the objectives for quality improvement and plans to achieve these objectives must be identified.
- Section 7: Support - The support section concerns the management of all resources related to the QMS and illustrates the need to control all resources, including human resources, buildings and infrastructures, work environment, monitoring resources and organizational measurement and knowledge. The section also includes requirements relating to the competence, awareness, communication and control of documented information (the documents and records required for the processes).
- Section 8: Operation - Operational requirements cover all aspects of planning and creating the product or service. This section contains requirements related to planning, reviewing product requirements, designing, auditing external suppliers, creating, and distributing the product or service, and checking for non-compliant process results.
- Section 9: Performance Evaluation - This section includes the requirements necessary to ensure that you can monitor the proper functioning of your QMS. These requirements include process monitoring and measurement, customer satisfaction assessment, internal audits and management review of the QMS.
- Section 10: Improvement - This last section includes the requirements necessary to improve your QMS over time. This includes the need to assess process non-conformities and the adoption of corrective actions related to processes.

These sections are based on a PDCA (Plan-Do-Check-Act) cycle, which uses these elements to implement change within the organization's processes, in order to stimulate and maintain improvements within the processes.

In ODIN, the DevOps guidelines presented in the corresponding sub-sections in all steps of the DevOps workflow (Sections 2 to 9) are meant to be followed in order to ensure that a high level of quality is achieved in the development, delivery and deployment of the ODIN components. The guidelines are based on best practices in each corresponding area. Following these guidelines will ensure that all ODIN partners have a common understanding and framework for component development and sharing, thus facilitating the development procedure and minimizing problems during component delivery and maintenance.

Certain parts of the DevOps workflow can be formalized enough that automatic tests can be developed to check if the corresponding guidelines are followed. Such automatic tests will be used whenever possible, to ensure high quality delivered products. In case that automatic tests are not possible, manual checks or appropriate design principles will be used to ensure that the guidelines are followed and to facilitate the production of high quality results.

A list of quality assurance objectives with respect to DevOps functionalities is presented in Table 9. These objectives cover mostly source code quality and functionality testing. For each objective, a qualitative or quantitative target is specified, along with the automatic or manual means to check if the target is achieved. Operation-time quality assurance will be also monitored through the deployment and KPI monitoring tools of Section 7, but these are more related to the achievement of functional and non-function requirements, to be evaluated within the activities of WP7.

Table 9: Quality assurance objectives for DevOps.

Objective	Target	Means to check
Source code directory structure	Structure should be as described in Section 2.4.	Simple automatic tests can be implemented to check if all necessary files (`README.MD`, `LICENCE.TXT`, `Dockerfile` etc.) exist and are properly named.
Source code quality	Minimum bugs, code is clearly written, stylistic conventions followed.	Automatic static code analysis tools such as SonarQube[123] and linters, e.g. Pylint[124].
Source code documentation	All publicly exposed functions and services should be documented (functionality, input, output).	By design: Existing documentation management frameworks will be used by the developers to document the components (see Sections 2.4 and 9.2.2). Automatic tests will be implemented to check if all released API is properly documented.
Code coverage by tests	80% code coverage[125].	Use of automatic code coverage tools, such as Codecov[126], Clover[127], Cobertura[128] and Coverage.py[129].

[123] SonarQube, https://www.sonarqube.org/ Last access June 2021.

[124] Pylint, https://www.pylint.org/ Last access June 2021.

[125] It is generally advised that code coverage is not used as a strict target, but rather as an indication of points in the code that have not been tested. It should not be responsible for allocating much effort in creating tests that are of little value, only to increase the code coverage rate. See e.g. https://medium.com/@nicklee1/why-test-code-coverage-targets-are-a-bad-idea-1b9b8ef711ef (last access June 2021).

[126] Codecov, https://about.codecov.io/ Last access June 2021.

Timely delivery of components	Components are delivered according to planned deadlines (in the DoA) and bug	By design: Use of GitFlow[130] branching scheme for feature development and hotfixes (see Section 2.2.3.1).
Component versioning	Version numbers are incremented properly and no new features are added to already released versions.	By design: Use of semantic versioning (see Section 5.4) will be used for component versioning.
Unit/Integration test success	100% success of unit and integration tests.	Use of automatic and manual tests written by developers and integrators and run either automatically in DevOps pipelines or manually by testers through test reporting platforms (see Section 4.4).
Security	Minimum source code vulnerabilities.	Use of automatic code quality checking tools such as SonarQube.

[127] Atlassian Clover, https://www.atlassian.com/software/clover Last access June 2021.

[128] Cobertura, http://cobertura.github.io/cobertura/ Last access June 2021.

[129] Coverage.py, https://coverage.readthedocs.io/en/coverage-5.5/ Last access June 2021.

[130] GitFlow, https://datasift.github.io/gitflow/IntroducingGitFlow.html Last access June 2021.

10 Conclusion

In order to achieve continuous integration and continuous delivery for a large project such as ODIN, several steps and tools need to be considered. This deliverable attempted to organize and clarify the DevOps procedure, from source code management to deployment monitoring, providing guidelines to the technical partners of ODIN. At each DevOps aspect, the available tools were examined in order to select appropriate ones for the requirements of ODIN.

This deliverable is meant to be used as a reference point and guide for ODIN's technical team. The reader can consult the "guidelines" sections at each chapter, summarized in Appendix A, to quickly find the recommended procedures to ensure high quality component delivery, moving to other parts of the deliverable for more details, if needed.

The document and the guidelines are susceptible to changes as the project progresses, as the requirements evolve and as components are being developed and deployed. For this reason, and due to the lack of another version of this deliverable, the document will be transferred to the project's Wiki, once the latter is released, in order to serve as a live reference guide, updated when needed.

Appendix A DevOps service list

The following table summarises ODIN's DevOps infrastructure. The table lists each step of the DevOps workflow, including the URLs of the corresponding servers and links to the corresponding guideline sections in the deliverable. The table can be used as a reference map of the DevOps infrastructure, to guide developers and deployers throughout their involvement in the ODIN project. This reference will be included in the project's Wiki as a running reference, and will form the basis for the implementation of the DevOps home page.

Table 10: ODIN DevOps service list.

DevOps functionality	Server	URL	Guidelines
Access to all services	DevOps home page	https://dev.odin-smarthospitals.eu	Section 9.3
Source code management	GitLab	https://gitlab.odin-smarthospitals.eu	Section 2.4
Building and containerization	GitLab	https://gitlab.odin-smarthospitals.eu	Section 3.3
Testing	Testing infrastructure	To be decided during the course of the project	Section 4.4
Software release	Docker registry	https://registry.odin-smarthospitals.eu	Section 5.4
Deployment management	Portainer / Docker swarm / Kubernetes	https://portainer.odin-smarthospitals.eu	Section 6.3, Section 7.4
KPI monitoring	KPI monitoring tool	To be decided during the course of the project	Section 7.4
Feedback collection and ticketing	Ticketing server	To be decided during the course of the project	Section 7.4
CI/CD pipeline orchestration	Jenkins server	https://jenkins.odin-smarthospitals.eu	Section 8.2
DevOps authentication	Keycloak server	https://account.odin-smarthospitals.eu	Section 9.1.4
Public Key Infrastructure	SKS keyserver	https://sks.odin-smarthospitals.eu	Section 9.1.4
Documentation	ODIN knowledge base (wiki)	To be decided during the course of the project	Section 9.2.1

Appendix B Acronym glossary

Acronym	Definition
AI	Artificial Intelligence
CI/CD	Continuous Integration / Continuous Delivery
DDOS	Distributed Denial-Of-Service
DevOps	Development Operations
ECS	Amazon Elastic Container Service
GUI	Graphical User Interface
ICREA	Catalan Institution for Research and Advanced Studies
IDE	Integrated Development Environment
IPR	Intellectual Property Rights
ISO	International Organization for Standardization
IT	Information Technology
ITIL	Information Technology Infrastructure Library
KPI	Key Performance Indicator
MPOE	Minimum Point of Entry
MS	Milestone
MW	Mega-watts
NOC	Network Operations Centre
ROS	Robot Operating System
SCM	Source Code Management
SOC	Security Operations Centre
SSAE	Statement on Standards for Attestation Engagements
SVN	Apache Subversion
TI	Technical Infrastructure
UI	User Interface
UPS	Uninterruptible Power Supply
URL	Uniform Resource Locator

VESDA	Very Early Smoke Detection Air
VCS	Version Control System
WP	Work Package