Ensure Software Quality

Mario David, Miguel Colom, Daniel Garijo, Leyla Jael Castro, Violaine Louvet, Elisabetta Ronchieri, Massimo Torquati, Laura del Caño, Leong Cerlane, Maxime van den Bossche, et al.

▶ To cite this version:

Mario David, Miguel Colom, Daniel Garijo, Leyla Jael Castro, Violaine Louvet, et al.. Ensure Software Quality. European Comission. 2024, https://zenodo.org/records/10723608. hal-04485911

HAL Id: hal-04485911
https://hal.science/hal-04485911v1

Submitted on 1 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Ensure Software Quality

EOSC Task Force 5 Sub Group 3

David, Mario (LIP)
Colom, Miguel (ENS Paris-Saclay)
Garijo, Daniel (UPM)
Castro, Leyla Jael (ZB MED)
Louvet, Violaine (CNRS)
Ronchieri, Elisabetta (INFN/CNAF)
Torquati, Massimo (UNIPI)
del Caño, Laura (CSIC/CNB)
Leong, Cerlane (CSCS)
Van den Bossche, Maxime (KU Leuven)
Campos, Isabel (CSIC/IFCA)
Di Cosmo, Roberto (INRIA)

February 28, 2024

Contents

1 Introduction **1**
1.1 The Research Software lifecycle . . . 2
1.2 Document organization . . . 2

2 Classification of quality characteristics, attributes, and metrics **3**
2.1 Software Quality Models: survey . . . 4
2.2 Software Quality Characteristics . . . 5
2.3 Software Quality Attributes and Metrics . . . 8

3 Landscaping **10**
3.1 Definition and references of Research Software . . . 10
3.2 Research Software stack . . . 10
3.3 Examples for Research Software stack and types . . . 11
3.3.1 By Research Software type . . . 11
3.3.2 Layer by layer in the software stack . . . 12

4 Recommendations **13**
4.1 Quality Attributes recommendations . . . 13
4.1.1 Common recommendations . . . 13
4.1.2 User story: Individual development . . . 13
4.1.3 User story: Team Project . . . 13
4.1.4 User story: Team OSS . . . 14
4.1.5 User story: Team Service . . . 15
4.2 Example of tools, services and infrastructures . . . 16
4.2.1 About version control and software forges . . . 16
4.2.2 About Continuous Integration/Continuous Delivery - CI/CD . . . 17
4.2.3 About intellectual property and licensing . . . 17
4.2.4 About packaging . . . 18

4.2.5 About documentation . . . 18
4.2.6 About management bugs . . . 18
4.2.7 About tests . . . 19
4.2.8 About support . . . 20
4.2.9 About code analysis . . . 20
4.2.10 About security . . . 20
4.2.11 Transversal needs . . . 21

5 Perspectives 22
5.1 Developers . . . 22
5.2 Users . . . 24
5.3 Service providers . . . 24

6 Metadata for quality software 27
6.1 Metadata schemas for Software . . . 27
6.2 The FAIR principles for Research Software (FAIR4RS) . . . 28
6.3 Mapping Quality Attributes to FAIR4RS . . . 29

7 Summary and conclusions 30

A Quality Attributes 32
A.1 Source Code Metrics: EOSC-SCMet . . . 32
A.2 Time and Performance Metrics: EOSC-TMet . . . 33
A.3 Qualitative Attributes: EOSC-Qual . . . 34
A.4 DevOps-SW Release and Management Attributes: EOSC-SWRelMan . . . 37
A.5 DevOps - Testing Attributes: EOSC-SWTest . . . 40
A.6 Service Operability Attributes: EOSC-SrvOps . . . 42

1 Introduction

Quality assessment is an important trait for software and for services. It allows researchers and service managers to have higher trust that, during its use and operation, the software and related services will work as expected, give the correct results and meet their requirements. Furthermore, it also contributes to their maintainability, stability and sustainability while facilitating the collaboration between developers and promoting good practices of software development [1].

In the framework of the EOSC Association several Task Forces have been created with the following topics:

1. Implementation of EOSC.
2. Metadata and data quality.
3. Research careers and curricula.
4. Sustaining EOSC.
5. Technical challenges on EOSC.

Regarding topic 5, it was divided into three subtopics:

1. Technical interoperability of data and services.
2. Infrastructure for quality Research Software.
3. AAI Architecture.

Regarding the Task Force **"Infrastructure for Quality Research Software"**, it was further subdivided into 3 sub-groups:

1. Software Lifecycle (*sub-group 1*).
2. Information Science (*sub-group 2*).
3. Ensuring Software Quality (*sub-group 3*).

The present document regards the work performed by *sub-group 3* **Software Quality** in research.

It is deemed important to have a definition of Research Software (**RS** from here on). The sub-group has adopted the following definition [2]:

"Research Software (RS) is commonly used to refer to software used and/or generated in a research context, including and not limited to scientific, non-scientific, commercial, academic and non-academic research."

The main objectives of this report are:

- To provide technical and organisational recommendations to improve RS quality based on a thorough review of the literature, in particular the software used in the services offered through EOSC.
- Identify Quality Attributes that are appropriate for RS and make recommendations for RS based on a set of such Quality Attributes.

1.1 The Research Software lifecycle

The Research Software lifecycle has been the work of *sub-group 1* of this Task Force and is reported here [3]. This document reproduces the diagram show in Figure 1 due to its relevance regarding both the Quality Attributes related to the development phase (center box number 3), the Quality Attributes related to the Software release and management (boxes "Maintenance" and "Deployment" number 5) and the Quality Attributes related to FAIR for Research Software (box "Publication" number 4).

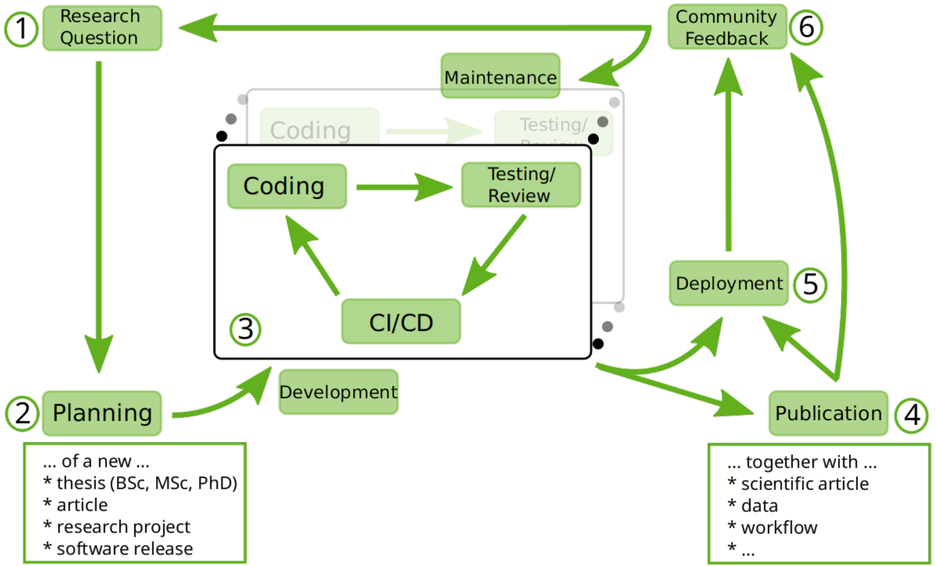


Figure 1: Research Software lifecycle. From *sub-group 1* [3].

1.2 Document organization

The structure of the document is as follows:

- Section 2 describes a survey conducted within the sub-group. The objective was to search for articles and documents describing Quality Models and corresponding Quality Attributes or Quality Characteristics. The complete list of Quality Attributes are given in Appendix A.
- In Section 3, the RS stacks, RS types and user stories, are identified. This classification is important for the recommendations in next section.
- In Section 4, recommendations are given for what Software Quality Attributes are more relevant for RS, depending on the stack, type and users or teams in their role as developers. There are also recommendations for software release and management and for services/platforms in production.
- In Section 5, the user perspectives are described, i.e., the point of view of developers, users and resource (or service) providers.
- Section 6 associates Quality Attributes and FAIR principles for RS.
- Last, the summary and conclusions are drawn in Section 7.

2 Classification of quality characteristics, attributes, and metrics

A significant notion in software engineering is *quality*, understood as a series of desirable particularities that are required for improved computer programs, meaning that they are well adapted to their purpose, accurate, highly available, or any other property which makes them more robust.

Whereas this is a concept which is easy to understand, arriving at a consensus on how to properly define it, is a significantly more complex task. Indeed, the problem can be addressed from plenty of different perspectives, each of them giving more or less importance to some particularities.

It is not the goal of this report to propose any particular definitions, but to analyse how the problem has been addressed in relevant works in the literature and how the characteristics can be classified as attributes. As such, in Section 2.1 we detail the method used for a survey of the literature.

In Section 2.2 we review what are, according to the literature, the *Quality Characteristics* that can be related to good software. The characteristics can be related to quality from very different points of view (ranging from metrics of the source code to service operability, just as an example). These are classified as *Quality Attributes and Metrics* in Section 2.3.

It is quite important to define the notions of **Verification and Validation** (**V&V**) of the *Quality Attributes and Metrics*, as well as **Test**, since this is the cornerstone to prove the correctness and user requirements satisfaction of a given Software or service. The following definitions from 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology [4] apply:

- Software Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.
- Software Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of the respective phase.
- Software Test: An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation i s made of some aspect of the system or component.

Although reproducibility does not explicitly appear as a characteristic of the software itself in our survey, it is a fundamental requirement for Research Software. See Section 3.1 for our definitions and context of *Research Software*. Indeed, the Scientific Method is, after all, based on the possibility of being able to repeat the experiments and refuting them to build up new knowledge. Specifically for Research Software, we need to consider its reproducibility as a characteristic of quality.

The definitions of repeatability, reproducibility, and replicability depend much on each research field. Sometimes the concepts of reproducibility and replicability have their meaning swapped with respect to other communities or disciplines.

Some consensus seems to have been reached in computational sciences after the report *Artefact Review and Badging*[1] version 1.1 by ACM. The scope of their definitions is intended to be very wide and to cover as many disciplines as possible. They refer, in a generic way, to the *team* and their *experimental setup*.

[1] `https://www.acm.org/publications/policies/artifact-review-and-badging-current`

We shall use these definitions in this report and put them in the context of computer science with the following meaning:

- **Repeatability**: Performed with the same team and same experimental setup. In our context, it means that the software can be executed as many times as needed. Researchers should be able to run the program several times always obtaining the same results.
- **Reproducibility**: Performed by a different team but the same experimental setup. Regarding computing, we could think of two groups of developers which need to obtain results from a program given its source code. If the two groups have exactly the same computer hardware and the same environment, they could simply compile the same source code, obtain the same executable binary code, execute the program, and obtain the same (or equivalent) results when providing the program with the same input data.
- **Replicability**: Performed by a different team and with a different experimental setup. In our context and following the previous example, one of the groups do not have access to the source code of the research method, but instead they have the corresponding detailed description published in a journal. These detailed descriptions can be, for example, a pseudocode or text in the article where all the parameters and significant operations are completely specified. Thus, replicability is defined as their capacity to arrive at an equivalent program after following the descriptions. Lack of replicability could be explained in this example due to missing pre or post-processing steps, missing parameters or their values, or ambiguous parts left for the interpretation of the reader.

In Section 5.2 we discuss the users' perspectives regarding reproducibility of Research Software.

2.1 Software Quality Models: survey

The classification into characteristics and attributes is based on a thorough review of the literature, in order to avoid biases on the classification and recommendations with the personal preferences or pre-established beliefs of the members of the *sub-group*. We followed the protocol and methodology proposed by Kitchenham and Charters [5] consisting of the following steps:

1. **Source selection and search**: We searched in the SCOPUS database, including the top five journals in software engineering related to software [2] and articles of the "International Conference on Software Engineering", one of the top venues for software engineering. We also added documents and web resources that the Task Force *sub-group* considered relevant. The search included the keywords *"software quality"* in the title of the publications. The following journals were considered:
    - IEEE Transactions on Software Engineering.
    - Empirical Software Engineering.
    - Journal of Systems and Software.
    - Software & Systems Modeling.
    - Information and Software Technology.
    - IEEE Software.
    - Software Quality Journal.

[2] `https://research.com/journals-rankings/computer-science/software-programming`

The following SQL query was used to filter out the articles in this step:

```
TITLE ( software  AND quality )  AND
    ( LIMIT-TO ( EXACTSRCTITLE ,  "Software Quality Journal" )
    OR  LIMIT-TO ( EXACTSRCTITLE ,  "Proceedings International Conference On Software Engineering" )
    OR  LIMIT-TO ( EXACTSRCTITLE ,  "IEEE Transactions on Software Engineering" )
    OR  LIMIT-TO ( EXACTSRCTITLE ,  "Empirical Software Engineering" )
    OR  LIMIT-TO ( EXACTSRCTITLE ,  "Journal of Systems and Software" )
    OR  LIMIT-TO ( EXACTSRCTITLE ,  "Software & Systems Modeling" )
    OR  LIMIT-TO ( EXACTSRCTITLE ,  "Information and Software Technology" )
    OR  LIMIT-TO ( EXACTSRCTITLE ,  "IEEE Software" )
    )  AND  ( LIMIT-TO ( SUBJAREA ,  "COMP" )  OR  LIMIT-TO ( SUBJAREA ,  "ENGI" ) )
```

We obtained 272 results with this process. Additional filtering was applied with the following criteria:

- Articles with no abstracts.
- Articles which were simple summaries of already existing proceedings.
- Articles that a preliminary review of the abstract and title made clear that were out of our scope.
- Articles that did not propose any quality dimensions. For example, those papers that just discuss practices.

2. **Inclusion and exclusion criteria**: The most important criterion for not considering articles in our survey was to exclude those which simply did not belong to the software engineering field. We did not attempt to tighten much the criteria for the exclusion, as we intended to gather as much related material as possible to be filtered out in the next field. Another early exclusion criterion was the language; we excluded non-English articles.

3. **Selection procedure**: After the previous selection, 147 papers remained for further analyses by reading their title and abstract. This allowed us to assess the pertinence of the article to the software engineering domain and their relevance to be used as a source for general quality software recommendations, assuming it mentioned software quality attributes. Although the protocol that we followed ensured that no human bias was introduced, it also prevented some very relevant papers related to Quality from being considered. Four additional articles were added in this step [6, 7, 8, 9], after the *sub-group* discussed the issue.

4. **Review process**: After the previous selection procedure, 19 relevant articles remained. These are the ones which are finally reviewed in our survey. We performed this step by groups of 2 or 3 reviewers per article, in our *sub-group*.

The list of relevant articles obtained by this procedure, allowed us to define a list of characteristics and attributes which are presented in the following sections.

2.2 Software Quality Characteristics

We identified a list of significant quality characteristics mainly from three sources:

- The ISO/IEC 25010:2011(E) standards [10]; is a norm that proposes two models for the characteristics. The first group is related to the context in which the software product is used, and contains five characteristics. The second group has eight characteristics according to characteristics of the software of the computer system itself, without relying on how it is used.
- The ISO/IEC/IEEE 24765:2017 standard [11]; defines a common vocabulary for systems and software engineering, in a way that it is always applicable to general applications.

This standardised list provides a vast quantity of very precise definitions that avoid trivial misinterpretations, and that therefore we apply in this report.

- The chapter *Design Fundamentals* of the Microsoft Application Architecture Guide [12]; the chapter *Quality Attributes* from the Microsoft's Application Architecture Guide also helped establish a common terminology in our definitions.

The analysis of the existing literature resulted in a significant number of characteristics which were associated with the concept of quality. All the characteristics were taken into account in our study, but some of them were filtered out since they were not useful to perform the classification in quality attributes. Indeed, some characteristics were pertinent at the time of the publication of the article, but after a few years they became obsolete. For example, producing a very small sized compiled binary *per se* as an indication of quality does not make much sense nowadays, whereas in the past it could be directly related to the ability of storing the program in a very limited memory or permanent media. Other characteristics were discarded because they were very specific to a given domain, such as real-time applications or critical systems. They remain valid characteristics to take into account, but they are not general enough or not applicable to Research Software or useful to the proposed recommendations.

We ended up with a list of 25 significant quality characteristics from the three above mentioned sources. The characteristics are defined as follows:

1. **Accessibility**; degree to which a product or system can be used by people with the widest range of diversity and capabilities to achieve a specified goal in a concrete context of use [10]. The range of capabilities includes diversity associated with age or any functional diversity. Accessibility can be specified or measured either as the extent to which a product or system can be used by users with diverse capabilities to achieve specified goals with effectiveness, efficiency, freedom from risk and satisfaction in a specific context of use, or by the presence of properties which specifically support the product's accessibility.

2. **Attractiveness**; degree to which a user interface enables pleasing and satisfying interaction with the user [10]. For example, one can include here properties of the product or system such as the use of colour and the nature of the graphical design. This characteristic is also known as *interface aesthetics.*

3. **Availability**; degree to which a system, product, or component is operational and accessible when required for use. We adapted this definition from [11].

4. **Confidentiality**; degree to which a product or system ensures that data are accessible only to those who have been authorised access [10]. This characteristic is a Security characteristic specialised on the privacy of the data.

5. **Compatibility**; degree to which a product, system, or component can exchange information with other products, systems or components, and performs its designed functions, sharing the same hardware or software environment [10]. This definition is very close to the one of *Interoperability.* Here we focus on the ability of the software to perform its functions in a suitable environment, whereas in Interoperability we refer to the format of the data which is exchanged.

6. **Ease of use (or *usability*)**; degree to which a product or system can be used by particular users to achieve their own goals with effectiveness, efficiency, and satisfaction in a given context of use. Usability can either be specified or measured as a product quality characteristic in terms of its sub-characteristics, or specified or measured directly by a subset of

System/Software Product Quality [10].

7. **Fault tolerance**; degree to which a system, product, or component operates as intended despite the presence of hardware or software faults. We adapted this definition from [11].
8. **Functional suitability**; degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions [10].
9. **Installability**; degree of effectiveness and efficiency a product or system can be successfully added or removed in a specified environment [10].
10. **Interoperability**; degree to which two or more systems, products, or components can share data, encoded in agreed formats. This definition was adapted from [11].
11. **Maintainability**; degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers [10]. Modifications can include corrections, improvements, or adaptation of the software to changes in the environment, and in the requirements and functional specifications. Modifications include both those carried out by specialised support staff, as well as those carried out by business or operational staff or end users. Maintainability can be interpreted as either an inherent capability of the product or system to facilitate maintenance activities, or the quality-in-use experienced by the maintainers for the goal of maintaining the product or system. It includes installation of updates and upgrades.
12. **Modifiability**; degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading the product [10].
13. **Operability** and **Manageability**; degree to which a product or system has attributes that make it easy to operate and control [10]. *Operability* corresponds to *Controllability*, that is, the operator's error tolerance and the conformity with users' expectations.
14. **Performance**; related to how well the system works taking into account the amount of resources used under stated conditions [10]. The *resources* might include other software products, the software and hardware configuration of the system, and any needed supplies (as for example print paper or storage media).
15. **Portability / Adaptability**; degree of effectiveness and efficiency with which a system, product, or component can be transferred from one hardware, software or other operational or usage environment to a different one [10]. Portability can be interpreted as either an inherent capability of the product or system to facilitate porting activities, or the quality-in-use experienced for the goal of porting the product or system.
16. **Recoverability**; degree to which, in the event of an interruption or a failure, a product or system can recover the directly affected data and re-establish the desired state of the system [10]. Following a failure, a computer system will typically be down for some time, which is determined by its *recoverability*.
17. **Reliability**; degree to which a system, product, or component provides specific functions under concrete conditions for a specified period of time. This definition is adapted from [11]. Lack of reliability can sometimes be caused by flaws in the requirements, the design, and the implementation, as well as because of contextual changes.
18. **Resource utilisation**; degree to which the amount and types of resources consumed by a product or system when performing its functions, meet the requirements [10]. Human resources are included in this category.

19. **Reusability**; degree to which a software artefacts (say, code, executable files, or any assets) can be exploited in other systems, or utilised to build other artefacts [10].
20. **Safety**; degree to which a product or system mitigates the potential personal risk to humans or to system components, in the intended contexts of use [10].
21. **Scalability**; this characteristic can be seen from the point of view of the system, or the applications. In the first case it is the measure of a system's ability to increase or decrease in performance and cost in response to changes in application and system processing demands. Examples would include how well a hardware system performs when the number of users is increased, how well a database withstands growing numbers of queries, or how well an operating system performs on different classes of hardware. Enterprises that are growing rapidly should pay special attention to scalability when evaluating hardware and software [13]. When referring to algorithms, protocols, or applications it can be defined as being able to efficiently handle a growing demand of work or need of more performance by means of adding more resources to the system on which the software is running. Resources can be added both to single nodes (vertical scalability) and to the system as a whole (horizontal scalability) [14].
22. **Security**; degree to which a product or system protects the information and data it manages (say, stores or transmits) in a way that access is only given to persons or systems with the appropriate level of authorisation they were granted [10]. Related to Security we can also find:
    - **Survivability**; degree to which a product or system continues to fulfil its mission by providing essential services in a timely manner despite the presence of attacks, covered by **Recoverability**.
    - **Immunity**; degree to which a product or system is resistant to certain attacks, covered by **Integrity**.
23. **Supportability**; is it defined as the ability of the system to provide helpful information to identify and resolve issues in case of malfunction [12]. The existence of a helpdesk, issue tracking, bug reporting, or related services also contribute to supportability [7].
24. **Testability**; degree of effectiveness and efficiency with which test criteria can be established for a system, product, or component. Then the tests might be run to determine whether the criteria have been met [10].
25. **Time behaviour**; degree to which the response, processing times, and throughput rates of a product or system meet the requirements when performing its functions [10].

2.3 Software Quality Attributes and Metrics

The *Software Quality Characteristics* which are classified in the previous section can actually be seen from very different points of view, thus allowing for a different type of classification of characteristics in different classes, which we will refer to as *Quality Attributes and Metrics*.

The survey and procedure described in Section 2.1 allowed us to identify 239 *Quality Attributes and Metrics* that were gathered into a single table. We found that, depending on the source, the same attribute or metric could be found under different names, but with a very similar definition or meaning. Those were merged to prevent duplication.

We ended up with 126 *Quality Attributes and Metrics*, it includes the references to the original articles used. We considered attributes which objectively could be treated as *metrics*. The attributes were subdivided into six categories and, for each metric or attribute, a new codename was proposed in order to have a coherent naming convention throughout the document. The following categories and their codenames are proposed:

- Source Code Metrics (**EOSC-SCMet**): 17 metrics. Metrics related to the source code, such as the number of lines of code or the number of assertions, for example.
- Time and Performance Metrics (**EOSC-TMet**): 11 metrics. Metrics related to time or periods of time. For example, the number of resolved bugs per period of time.
- Qualitative Attributes (**EOSC-Qual**): 27 attributes. Qualitative attributes are obtained in general through surveys to, or some manual analysis by software developers, administrators, or users. In general, these are not possible to automate and are in general subjective.
- DevOps - Software Release and Management Attributes (**EOSC-SWRelMan**): 34 attributes. Largely based on the *DevOps* methodology, they can be automated for Verification & Validation. Although the possibility of having code reviews in the software development process is a manual step.
- DevOps - Testing Attributes (**EOSC-SWTest**): 25 attributes. Again based in DevOps, these are related to software testing and can also be automatically verified. For example, whether integration tests are used in the system.
- Service Operability Attributes (**EOSC-SrvOps**): 12 attributes. They refer to scientific services or platforms in operation. For example, whether the system provides monitoring and accounting services.

In Appendix A each attribute or metric entry has the following elements, as shown in Figure 2:

- Codename: naming convention proposed by the *sub-group*;
- Name: as found in the source reference;
- Associated characteristics: one or more from subsection 2.2;
- Definition: obtained from aggregating or merging the source references;
- Research Software level: from the user stories in subsection 3.1;
- Reference to the source articles.

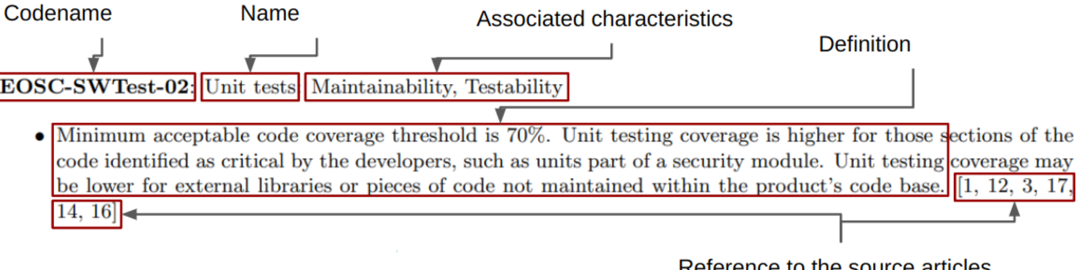


Figure 2: The structure of each Attribute or Metric entry, as it appears in the Appendix.

3 Landscaping

3.1 Definition and references of Research Software

Research Software is not only software. To be able to clearly identify quality attributes for Research Software we rely on its definition given in page 1.

The most important characteristic is the link with scientific publications. The purpose of Research Software is that it supports the scientific results, and from that it is expected to reproduce the published results. Regular software does not necessarily have this expectation.

The goal of this section is to cross-reference the elements of software quality from the previous section with the *sub-group 1* [3] ("Software Lifecycle", c.f. Section 1), typology of Research Software development, in order to identify the tools and infrastructures needed to satisfy minimum quality criteria, taking into account the specificities of Research Software.

As a reminder, *sub-group 1* has defined different categories of software development related to the context in which they are performed:

1. **Individual development** - Individual creating Research Software for his/her own use (e.g. a PhD student): Based on a research question, software is created by a single person with the specific aim of answering the research question and producing research output (paper, dataset, etc).
2. **Team Project** - A research team creating an application or workflow for the use within the team: Research Software is created by a team to answer a series of research questions (often as part of a larger research project).
3. **Team OSS** - A team / community developing (possibly broadly applicable) open source Research Software: Software is created by a team (possibly distributed over multiple organisations) to answer a broad range of research questions.
4. **Team Service** - A team or community creating a research service: A service platform is a set of software components which are used to provide services for a large number of users, most of whom make use of these offerings via the Internet (e.g. cloud services).

Although there are other references, for instance [3] define similar or equivalent categories as enumerated above while [4] rely on the same definition of RS as we did in page 1.

3.2 Research Software stack

A typical scientific software stack [15] has the composition shown in Table 1.

Note that Stack level 1 is not considered in this report. The Research Software types are summarised next including some examples:

1. **Library**: SciPy, TensorFlow, FFTW, BLAS, LAPACK.
2. **Framework**: JupyterLab, RStudio.
3. **Application** (such as Monte-Carlo simulation Software): Gromacs, Amber, GEANT4, COMSOL.
4. **Analysis script and workflows**: Jupyter notebooks, R scripts.

[3] https://docs.google.com/presentation/d/1uwxSwd8chbG7bVn5lPvNNhv5f_JeeelA2mN9NXhlOhA
[4] https://zenodo.org/record/7589725

5. **Services and platforms**: Galaxy server, Scipion, SAPS, O3AS.

Stack	Stack definition	Types
4 - Project specific code	Software written by scientists for a specific research project. It can take various forms including scripts, notebooks, and workflows, but also special-purpose libraries and utilities.	Library; Analysis script and workflows; Services and platforms
3 - Domain specific tools	Tools and libraries that implement models and methods which are developed and used by specific communities. GROMACS, MMTK, Amber.	Library; Application; Services and platforms
2 - Scientific infrastructure	Infrastructure created specifically for scientific computing, but not for any particular application domain: mathematical libraries such as BLAS, LAPACK, or SciPy, scientific data management tools such as HDF5.	Library; Framework; Services and platforms
1 - Non-scientific infrastructure	*Compilers and interpreters, libraries for data management; gcc, python...*	–

Table 1: RS stacks, stacks definitions and RS types. Stack level 1 is not considered in this report.

3.3 Examples for Research Software stack and types

To be able to identify quality attributes for Research Software, we need to understand clearly what is expected from the different types identified in the previous sections.

3.3.1 By Research Software type

The different expectations described below add up to each other, when the context of development grow in complexity.

Individual researcher creating software for personal use

The author must be able to manage the different versions of the code. He/She often has to come back to the software after a long time (to answer to a reviewer for a publication, to address a new scientific question or confirm previous results).

He/She perhaps needs to execute the software on different computers and different operating systems or versions (for example the use of a supercomputer).

He/She has to share the code when he/she submits his article.

A research team creating an application or workflow for use within the team

The team must be able to manage the development with several people. The way to program should be homogeneous, and everybody must have a clear vision of who does what.

A new arrival in the team must be able to easily understand the structure of the software and the way to develop its code.

A team / community developing Open Source Software

For a community Research Software, it is important to facilitate different types of contribution and to manage correctly the rights attribution.

It is important to take into account Validation & Verification of the code. The code has to be easy to maintain and to upgrade.

A community software must be as easy as possible to install, use and report bugs, problems or suggestions/new features. Strong software support must be in place to help the users.

A team or community creating a service, a platform or an infrastructure

The main expectations are availability, reliability, security, performance and scalability of the service or platform for users and operators.

A team / community developing software in an industrial context

In the case of industrial context, expectations can vary depending on the contract between academic researchers and the company.

3.3.2 Layer by layer in the software stack

Regarding this categorisation, the user's point of view supersedes the developer's point of view.

A scientific infrastructure

A scientific infrastructure, as described in [15], is by definition a software used by other software. That is, it is not a standalone code. Therefore it must be easy to install and to interface/integrate with other software.

Users expect this type of software to be robust and to be sufficiently sustainable since it is often the foundation of their own code.

A domain specific tool

A domain specific tool is similar to that of a community software. It must be easy to install on different operating systems and easy to use. It should have good retro-compatibility. It should be easy to cite. It is important for users to be able to know precisely what the software does without having to read the source.

A project specific code

A project specific code has similar characteristics and requirements to that of a single person or team developing the code. Reproducibility plays an important role. It must be easily citable and archived for sustainability reasons. It often depends on the scientific infrastructures.

4 Recommendations

4.1 Quality Attributes recommendations

4.1.1 Common recommendations

Table 2 shows the list of Quality Attributes that are common to all types of Software, and to all user stories described in subsection 3.1. They are deemed important and not too difficult to implement even by an individual researcher or software developer.

These attributes are about documentation, open source code including a license, visible description about intellectual property that is important to establish ownership and scientific recognition. They also include avoiding redundant code and addressing security concerns (e.g. no world writable files and directories).

RS Stack	RS Type
All	**All**
EOSC-SCMet-02: % of redundant code	
EOSC-SCMet-10: Number of comments	
EOSC-Qual-27: Intellectual Property	
EOSC-SWRelMan-01: Open source	
EOSC-SWRelMan-02: Version Control System (VCS)	
EOSC-SWRelMan-10: Open-source license	
EOSC-SWRelMan-29: Documentation online	
EOSC-SWRelMan-30: Documentation updates	
EOSC-SWRelMan-32: Documentation production	
EOSC-SWTest-15: No world-writable files or directories	

Table 2: Recommended common Quality Attributes.

4.1.2 User story: Individual development

In the case of individual development of analysis script and workflows (Table 3), the recommendations are the common attributes listed in Table 2.

RS Stack	RS Type
4: **Project specific code**	4: **Analysis script and workflows**
All common Quality Attributes: Table 2	

Table 3: Recommended Quality Attributes, individual development, Analysis script and workflows.

In the case of software type "Library", we add packaging and code deployment to the common attributes as listed in Table 4 (c.f. RS lifecycle diagram 1 box 5).

RS Stack	RS Type
4: **Project specific code**	1: **Library**
All common Quality Attributes: Table 2	
EOSC-SWRelMan-12: Packaging	
EOSC-SWRelMan-15: Code deployment	

Table 4: Recommended Quality Attributes, individual development, library.

4.1.3 User story: Team Project

In the case of a team working on a project on analysis script and workflows, there is an increase in responsibility. As such it is recommended to add support that includes the reporting and

solving bugs and source code hosting in common VCS repositories, as shown in Table 5. See RS lifecycle diagram 1 boxes *Maintenance* and *Community Feedback*.

RS Stack	RS Type
4: **Project specific code**	4: **Analysis script and workflows**
All from User story: Individual development: Table 3	
EOSC-TMet-02: # Resolved bugs	
EOSC-TMet-03: # Open bugs	
EOSC-SWRelMan-03: Source code hosting	
EOSC-SWRelMan-04: Working state version	
EOSC-SWRelMan-07: Support	

Table 5: Recommended Quality Attributes, team project, analysis script and workflows.

The case of Research Software type "Library", the recommendation is to add testing with respect to the previous cases, this is shown in table 6. See RS lifecycle diagram 1 boxes *Maintenance*, *Community Feedback* and *Testing/CICD*.

RS Stack	RS Type
4: **Project specific code**	1: **Library**
All from User story: Individual development: Table 4	
EOSC-TMet-02: # Resolved bugs	
EOSC-TMet-03: # Open bugs	
EOSC-SWRelMan-03: Source code hosting	
EOSC-SWRelMan-04: Working state version	
EOSC-SWRelMan-07: Support	
EOSC-SWTest-01: Code style	
EOSC-SWTest-02: Unit tests	
EOSC-SWTest-03: Test doubles	
EOSC-SWTest-07: Functional testing	

Table 6: Recommended Quality Attributes, team project, library.

Table 7 regards the development, deployment and operation of a service. We add recommendations related to further types of tests such as APIs and integration (when applicable). Furthermore, there are several Quality Attributes related to security such as the use of service certificates to encrypt endpoints, use of strong authentication and authorization mechanisms even in the case of limited access to the services. See RS lifecycle diagram 1 boxes *Maintenance*, *Community Feedback* and *Testing/CICD*.

RS Stack	RS Type
4: **Project specific code**	5: **Services and platforms**
All from User story: team project, library: Table 6	
EOSC-SWTest-05: API testing	
EOSC-SWTest-06: Integration testing	
EOSC-SWTest-16: Public endpoints and APIs encrypted	
EOSC-SWTest-17: Strong ciphers	
EOSC-SWTest-18: Authentication and Authorization	
EOSC-SWTest-20: Service compliance with data regulations (GDPR)	

Table 7: Recommended Quality Attributes, team project, services and platforms.

4.1.4 User story: Team OSS

This subsection describes the recommendations when a medium to large team is responsible for the development of Open Source Software for research. Thus, the increase in responsibility with respect to the previous cases of smaller teams or individual researchers, implies higher scores in the Quality Attributes. As such, we add Quality Attributes for more complete documentation,

additional types of tests including security and peer code review, as shown in Table 8. See RS lifecycle diagram 1 box *Testing/CICD*.

RS Stack	RS Type
3 and 2: **Domain specific tools** and **Scientific infrastructure**	1: **Library**
All from User story: Team Project: Table 6	
EOSC-TMet-04: Defect rates	
EOSC-SWRelMan-08: Code review	
EOSC-SWRelMan-26: Documentation version controlled	
EOSC-SWRelMan-27: Documentation as code	
EOSC-SWRelMan-28: Documentation formats	
EOSC-SWTest-08: Performance testing	
EOSC-SWTest-13: Static Application Security Testing (SAST)	
EOSC-SWTest-14: Security code reviews	

Table 8: Recommended Quality Attributes, team OSS, library.

Table 9 shows the case for a Framework or Application, it adds further types of testing such as scalability and development methodology called Test-Driven Development (TDD). See RS lifecycle diagram 1 box *Testing/CICD*.

RS Stack	RS Type
3 and 2: **Domain specific tools** and **Scientific infrastructure**	2 and 3: **Framework, Application**
All from User story: Team OSS, Library: Table 8	
EOSC-SWTest-04: Test-Driven Development (TDD)	
EOSC-SWTest-10: Scalability testing	

Table 9: Recommended Quality Attributes, team OSS, frameworks and applications.

The Quality Attributes related to software release management are shown in Table 10. In the case of an Open Source Software development team, software release and management is part of its responsibility. Thus, the additional recommended Quality Attributes include the existence of a helpdesk and/or bug tracking system, recommendation for software versioning, packaging and documentation. This reflects the RS lifecycle diagram 1 boxes *Maintenance*, *Community Feedback* and *Deployment*.

Quality Attribute
EOSC-SWRelMan-09: Semantic Versioning
EOSC-SWRelMan-13: Register/publish artefact
EOSC-SWRelMan-17: SCM tool
EOSC-SWRelMan-20: Installability
EOSC-SWRelMan-25: Provide checksums

Table 10: Recommended Quality Attributes for SW release and management.

4.1.5 User story: Team Service

This subsection refers to a medium to large team developing an Open Source scientific research service or platform that is used by many researchers, possibly from different fields. The list of Quality Attributes is significant with several types of tests including security, complete set of documentation, helpdesk for support and bug reporting. See RS lifecycle diagram 1 boxes *Maintenance*, *Community Feedback* and *Testing/CICD*.

In the case of services and platforms in production, additional Quality Attributes are recommended. Table 12 shows this list. They include the existence of a helpdesk and/or bug tracking system, monitoring, accounting, authentication and authorization mechanisms, documentation and policies to access and use the service or platform.

RS Stack	RS Type
3 and 2: **Domain specific tools** and **Scientific infrastructure**	5: **Services and platforms**
All from User story: Team Project: Table 7	
EOSC-TMet-04: Defect rates	
EOSC-SWRelMan-08: Code review	
EOSC-SWRelMan-26: Documentation version controlled	
EOSC-SWRelMan-27: Documentation as code	
EOSC-SWRelMan-28: Documentation formats	
EOSC-SWTest-04: Test-Driven Development (TDD)	
EOSC-SWTest-08: Performance testing	
EOSC-SWTest-09: Stress testing	
EOSC-SWTest-10: Scalability testing	
EOSC-SWTest-12: Open Web Application Security Project (OWASP)	
EOSC-SWTest-13: Static Application Security Testing (SAST)	
EOSC-SWTest-14: Security code reviews	
EOSC-SWTest-19: API security assessment	
EOSC-SWTest-21: Dynamic Application Security Testing (DAST)	
EOSC-SWTest-22: Interactive Application Security Testing (IAST)	
EOSC-SWTest-23: Security penetration testing	
EOSC-SWTest-24: Security assessment	
EOSC-SWTest-25: Security as Code (SaC) Testing	

Table 11: Recommended Quality Attributes, team service, services and platforms.

EOSC-TMet-08: # Registered users
EOSC-TMet-09: # Active users
EOSC-TMet-10: Amount computing resources
EOSC-TMet-11: Amount storage resources
EOSC-SWRelMan-17: SCM tool
EOSC-SWRelMan-20: Installability
EOSC-SWRelMan-22: Infrastructure as Code (IaC) validation
EOSC-SWTest-16: Public endpoints and APIs encrypted
EOSC-SWTest-17: Strong ciphers
EOSC-SWTest-18: Authentication and Authorization
EOSC-SWTest-20: Service compliance with data regulations (GDPR)
EOSC-SWTest-24: Security assessment
EOSC-SrvOps-01: Acceptable Usage Policy (AUP)
EOSC-SrvOps-02: Access Policy and Terms of Use
EOSC-SrvOps-03: Privacy policy
EOSC-SrvOps-04: Operational Level Agreement (OLA)
EOSC-SrvOps-05: Service Level Agreement (SLA)
EOSC-SrvOps-06: Monitoring service public endpoints
EOSC-SrvOps-07: Monitoring service public APIs
EOSC-SrvOps-09: Monitoring security public endpoints and APIs

Table 12: Recommended Quality Attributes for a service or platform in production.

4.2 Example of tools, services and infrastructures

4.2.1 About version control and software forges

Version control systems help track and manage changes to files and particularly to source code. A software forge is a web-based collaborative platform for developing and sharing codes.

The Quality Attributes related to version control are the following:

- EOSC-SWRelMan-02: Version Control System (VCS)
- EOSC-SWRelMan-03: Source code hosting
- EOSC-SWRelMan-04: Working state version

Version control is an essential tool for software development. Software forges have become vital tools for all software developers. This type of infrastructure is often deployed at different levels;

laboratory, project, institution, etc.. Commercial software forges (such as GitHub and GitLab), are significantly used in the academic context. A recent report of the Software and Sources Codes College of the Committee for Open Science illustrate the landscaping in France [16] shows this trend.

An important point is that most of the software forges integrate a large variety of tools (most of them listed below), that goes far beyond version control.

Software forges are a key platform to manage contributions to the code from many diverse people. They integrate the mechanism of Pull or Merge Requests, allowing any developer to contribute to the code and being peer reviewed, for the purpose of adding a new functionality, bug and security fixes, etc.

4.2.2 About Continuous Integration/Continuous Delivery - CI/CD

Continuous Integration and Continuous Delivery (CI/CD), is one of the best practices in software development that implement the DevOps methodology. Changes are immediately and automatically tested and executed, and packages (or artefacts) are built and published in public repositories.

CI/CD systems are the way to implement the verification of a set of Quality Attributes. The set of QAs to be verified, is implemented through a pipeline that is executed in one such CI/CD service. The CI/CD system is connected with the software forges, and the pipelines are triggered when there are changes in the source code. Package building, testing, deployment, documentation linting, are some of the tasks that can be automated in CI/CD pipelines.

Examples of CI/CD systems are services such as; Jenkins (`https://www.jenkins.io/`), GitLab CI (`https://docs.gitlab.com/ee/ci/`) and Travis CI(`https://travis-ci.com/`).

The Software Quality as a Service (SQAaaS) platform (`https://sqaaas.eosc-synergy.eu`), is a service developed in EOSC-Synergy project[5], it allows Jenkins pipeline composition for automatic tests as well as Badge awarding according to predefined verification of Quality Attributes and allows to perform Verification & Validation of the Software or Services.

4.2.3 About intellectual property and licensing

The quality attributes related to legal issues and licensing are:

- EOSC-Qual-27: Intellectual Property (IP)
- EOSC-SWRelMan-01: Open Source
- EOSC-SWRelMan-10: Open-Source license

There is no real tool to verify the IP, where support can be provided by the institution. This support must be as large as possible including the help about Developer Certificate of Origin (DCO), or Contributor License Agreement (CLA).

Regarding Open Source and license, there are tools to verify that a given Open Source license file is present in the software forge, comparing it with a know database of Open Source licenses, one such tool is "*licensee*"[6].

[5] `https://www.eosc-synergy.eu/`.
[6] `https://github.com/licensee/licensee`

4.2.4 About packaging

Software packaging is the process to build a self-contained package or artefact of a software, in order for it to be easily installable.

The Quality Attributes regarding this topic are:

- EOSC-SWRelMan-12: Packaging
- EOSC-SWRelMan-15: Code deployment

The number of packaging system is very large, thus we list just a few of such tools and services. We focus on mostly used tools in academic research and open source context. This is **not** an exhaustive list.

Tools	Usage	URL	Comment
Guix	Packaging system	`https://guix.gnu.org/en/`	Generate reproducible environment
Nix	Packaging system	`https://nixos.org/`	Generate reproducible environment
CMake	Build system	`https://cmake.org/`	Include many processes like compilation, packaging, testing
Docker	Container technology	`https://www.docker.com/`	Certainly the most used
Singularity	Container technology	`https://sylabs.io/`	Technology used in many computing centers
Kubernetes	Container orchestration	`https://kubernetes.io/`	Mostly used in services deployment
Openshift	PaaS	`https://www.redhat.com/fr/technologies/cloud-computing/openshift`	Based on Docker and Kubernetes technologies
RPM	Package manager	`https://rpm.org/`	RedHat and derivatives packaging system
deb	Package system	`https://wiki.debian.org/Packaging`	Debian/Ubuntu and derivatives packaging system

Table 13: Tools and services for packaging and deployment

It should be noted that each of these tools, services and packaging systems are available or used in most research infrastructures.

4.2.5 About documentation

Documentation is of critical interest for software and covers many Quality Attributes:

- EOSC-SWRelMan-29: Documentation online
- EOSC-SWRelMan-30: Documentation updates
- EOSC-SWRelMan-32: Documentation production
- EOSC-SWRelMan-26: Documentation version controlled
- EOSC-SWRelMan-27: Documentation as code
- EOSC-SWRelMan-28: Documentation formats

The production of documentation is mostly manual but there are quite a few number of tools that help generate or produce it.

4.2.6 About management bugs

Bug tracking systems or more generally issue tracking systems are common components for software development. The relevant Quality Attributes are:

Tools	Usage	URL	Comment
Doxygen	Automatic generation	`https://www.doxygen.nl/`	Multi-languages
Sphinx	Automatic generation	`https://www.sphinx-doc.org/en/master/`	For Python language
Javadoc	Automatic generation	`https://www.oracle.com/java/technologies/javase/javadoc-tool.html`	For Java language
MediaWiki	Wiki engine	`https://www.mediawiki.org/wiki/MediaWiki`	Available on GitLab platform
GitLab Pages	Automatic generation	`https://docs.gitlab.com/ee/user/project/pages/`	Available in GitLab platform with CI
Hugo	Static web site generation	`https://gohugo.io/`	Based on Go language
Pelican	Static web site generation	`https://getpelican.com/`	based on Python language

Table 14: Tools and services for documentation

- EOSC-TMet-02: # Resolved bugs
- EOSC-TMet-03: # Open bugs

Most software forges include a bug or issue tracking system. There are other tools such as Redmine (which is also a forge software like, `https://www.redmine.org/`), Trac (`http://trac.edgewall.org/`), GLPI (`https://www.glpi-project.org/`), Request Tracker (`https://bestpractical.com/request-tracker`) or Bugzilla (`https://www.bugzilla.org/`).

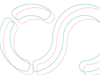
4.2.7 About tests

Automated Software testing is considered a best practice in software development. This can be implemented in CI/CD pipelines (Section 4.2.2) and covers the following Quality Attributes:

- EOSC-SWTest-02: Unit tests
- EOSC-SWTest-03: Test doubles
- EOSC-SWTest-07: Functional testing
- EOSC-SWTest-05: API testing
- EOSC-SWTest-06: Integration testing
- EOSC-SWTest-08: Performance testing
- EOSC-SWTest-13: Static Application Security Testing (SAST)
- EOSC-SWTest-04: Test-Driven Development (TDD)
- EOSC-SWTest-10: Scalability testing
- EOSC-SWTest-09: Stress testing

There are a large number of tools and they cover several programming languages and frameworks, we list below a few of them:

- For Python; tox (`https://tox.wiki`), pytest (`https://pytest.org/`).
- For Java; JUnit (`https://junit.org/`).
- For C++; CppUnit (`http://freedesktop.org/wiki/Software/cppunit`).



4.2.8 About support

The existence of a helpdesk is very important for users, operators, developers to create tickets about bugs, questions, new features, security issues, operation or usage of the software or service. The helpdesk contributes to the trust and adoption of the corresponding software or service. The Quality Attribute related to support is:

- EOSC-SWRelMan-07: Support

There are many services that can be used for support and we list a few of them:

- GitHub issues `https://docs.github.com/en/issues`: is an issue tracking system integrated within the GitHub repository
- GitLab issues `https://gitlab.com/gitlab-org/gitlab/-/issues`: is to GitHub but for GitLab.
- Jira from Atlassian `https://www.atlassian.com/software/jira`: is an issue and project tracking system.
- Request Tracker - RT `https://bestpractical.com/request-tracker`: Helpdesk.
- Zammad `https://zammad.com/en`: Ticketing system.

4.2.9 About code analysis

Static code analysis contribute to Software quality, the relevant Quality Attributes are the following:

- EOSC-SCMet-02: % of redundant code
- EOSC-SCMet-10: Number of comments
- EOSC-SWTest-01: Code style
- EOSC-SWTest-15: No world-writable files or directories
- EOSC-SWRelMan-08: Code review

There are many tools that can help to automate this analysis as part of CI/CD pipelines. These follow best practices and implement code conventions checking and well identified programming rules.

4.2.10 About security

Security is a critical topic especially for platforms and services, but also for other types of Software. Several Quality Attributes are related to security:

- EOSC-TMet-04: Defect rates
- EOSC-SWTest-12: Open Web Application Security Project (OWASP)
- EOSC-SWTest-13: Static Application Security Testing (SAST)
- EOSC-SWTest-14: Security code reviews
- EOSC-SWTest-16: Public endpoints and APIs encrypted
- EOSC-SWTest-17: Strong ciphers

Tools	Usage	URL	Comment
Pylint	Static code analysis	`https://pylint.pycqa.org/`	For Python language. Others *lint tools exists for others languages : CPPlint for CPP, JSLint for JavaScript
Hadolint	Dockerfile linter	`https://github.com/hadolint/hadolint`	To ensure best practice in Docker images
checkstyle	Static code analysis	`https://checkstyle.sourceforge.net/`	For Java language
pycodestyle	Static code analysis	`https://pypi.org/project/pycodestyle/`	Check Python code against some of the style conventions in PEP 8
flake8	Static code analysis	`https://flake8.pycqa.org/en/latest/#`	Check Python code against some of the style conventions in PEP 8
SonarQube	Automatic reviews with static analysis of code	`http://sonarqube.org/`	Supports many programming languages
gcov	Dynamic coverage analysis	`https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`	Program must be compiled with specific options
valgrind	Memory error detection	`https://www.valgrind.org/`	Runs programs on a virtual processor

Table 15: Tools for static and dynamic code analysis

- EOSC-SWTest-18: Authentication and Authorization
- EOSC-SWTest-19: API security assessment
- EOSC-SWTest-20: Service compliance with data regulations (GDPR)
- EOSC-SWTest-21: Dynamic Application Security Testing (DAST)
- EOSC-SWTest-22: Interactive Application Security Testing (IAST)
- EOSC-SWTest-23: Security penetration testing
- EOSC-SWTest-24: Security assessment
- EOSC-SWTest-25: Security as Code (SaC) Testing

Some of the analysis tools include dynamic security analysis. Others, such as Splint (`http://www.splint.org/`, for C language) or Bandit (`https://bandit.readthedocs.io/en/latest/` for Python language) can be used for static security code analysis.

4.2.11 Transversal needs

The tools and services listed in the previous sections are not enough to produce high quality software. Training is crucial at different level (students, researchers, and even software developers), in order to promote best practices and a good use of the different tools and services.

This can be done in different ways:

- Introduction of specific courses in formal education.
- Lifelong training.
- On line training (such as MOOCs)
- Seminars, webinars, tutorials, hands-on courses.

It is also of crucial importance to develop proximity support for all scientific communities on these topics.

5 Perspectives

Perspectives regarding Research Software quality include many different aspects that researchers and the scientific community at large consider relevant when evaluating the quality of software used or developed in a particular research field. These aspects are all equally essential to ensure the credibility and reliability of the research results themselves while meeting the expectations of software users. Clearly, the different perspectives on the quality of Research Software can branch significantly based on the individuals using it, the type of software, and the context in which it is used.

In the following, we shall discuss perspectives according to two distinct points of view: the one of the developers of Research Software, and that of the users who may be either other researchers extending research into the same or different research field or mere software users. Finally, we discuss the aspects related to the services and resources that enable the development, the execution, and the fruition of the results produced by Research Software.

5.1 Developers

Research software is developed by different individuals e.g., academic researchers, Research Software engineers (RSEs), as well as entities such as research departments in industries and academic laboratories, depending on the context and purpose of the software.

Section 3.1 defines several categories of software development related to the context in which they are performed. Here, we refer to the category of *Research Software developer*, which includes, but is not limited to, the following:

- **Academic Researchers and RSEs**: Researchers who develop software tools or applications to support their own research projects. Such software is often tailored to specific research needs and might be used extensively only in the research group or laboratory in which it originated. Increasingly, however, groups of academic researchers, in some cases with heterogeneous expertise, are collaborating within a laboratory or research project to build tools and software frameworks for their current research and to facilitate the study of other individuals or research groups. A virtuous example of Research Software that is widely used in academia and industry, resulting from the collaboration of European research groups with other international research players is Quantum ESPRESSO, a software suite for first-principles electronic-structure calculations and materials modeling, distributed under the GNU General Public License.
- **Research Institutions**: Research software is sometimes developed by research institutions, universities and research organizations (e.g., the CERN). These institutions often have dedicated software departments or laboratories that create software tools for various research projects.
- **Non-profit Organizations**: Some non-profit organizations (e.g., the Free Software Foundation) focus on the development of software and tools that are made available to the community (including the research community) free of charge or at a reduced cost, usually via donations.
- **Research communities**: Open research communities are essential for advancing and promoting collaborations in the research and scientific communities. These communities comprise individuals, organizations, and institutions committed to open research principles, including open access to research publications, data sharing, open-source software, and open

quality assurance assessment. Examples of research communities include the Center for Open Science (COS), the OpenAIRE (Open Access Infrastructure for Research in Europe) or thematic Research communities such as; WORSICA (Water Monitoring Sentinel Cloud Platform) or O3AS (Ozone Assessment Service).

Note that there is no direct relation between *Research Software developer* categories and software development categories, for example individual developer can initiate an OSS project. Furthermore, as observed by J. Cohen et al. [17] *good Research Software can make the difference between valid, sustainable, reproducible research outputs and short-lived, potentially unreliable or erroneous outputs.*

Research software developers are devoted to producing high-quality software that facilitates or enables scientific research and technological advancements but also adheres to best practices in software development to ensure reliability, maintainability, and long-term viability. Hence, regardless of the developer category, the quality and reliability of Research Software are essential to ensure the validity and reproducibility of research results.

One crucial aspect for Research Software developers is acknowledging the value of Research Software and recognising its authorship. This point may significantly impact the developers' progress in their careers. In many cases, the contributions of researchers who develop software are only acknowledged through citations in academic papers since Research Software is almost always directly or indirectly linked to one or more publications. In this regard the Coalition for Advancing Research Assessment (CoARA: `https://coara.eu/`), has the vision that the assessment of research, researchers and research organisations recognises the diverse outputs, practices and activities that maximise the quality and impact of research.

Additionally, traditional academic evaluation systems often underestimate the production of Research Software, preferring to rely on other achievements. Recently, many institutions have recognized the importance of addressing this issue, fostering a growing sensibility in establishing practices to assess and acknowledge Research Software development. Such new sensitivity will help create a more inclusive and accurate assessment of modern research specifically for what concerns Research Software development.

Another important aspect is the developers' rights regarding the Research Software they produce. Do they have all the rights to the Research Software they produce or contribute to develop? This aspect has several implications regarding intellectual property, exploitation or results, re-use in other research fields or contexts, re-licensing all or part of the software, and so on. The answer to the question depends on several factors, including institutional policies, funding agreements, collaboration agreements, and the licensing choices made by the developers for their software. Developers working within academic institutions or other research organizations may be subject to institutional policies regarding intellectual property. In some cases, the institution may claim ownership of the software or have specific policies governing the rights of developers.

Moreover, if the Research Software development is funded by external entities (such as government agencies, private organizations, or foundations), the funding agreement terms may stipulate the ownership and rights associated with the software. However, more and more often, some funding agencies encourage or require that software developed with their support be released as open-source. For example, the EU Open Source Software Strategy 2020-2023) states that *the internal strategy, under the theme "Think Open", sets out a vision for encouraging and leveraging the transformative, innovative and collaborative power of open source, its principles and development practices. It promotes the sharing and reuse of software solutions, knowledge and expertise, to deliver better European services that benefit society and lower costs to that society.*

Generally, developers retain copyright in their software unless institutional policies or employment contracts state otherwise. In that case, developers typically have the right to choose the license under which they release their software and thus determine how others can use, modify, and distribute it. In light of all these aspects, it is of foremost importance for developers to be aware of and understand the legal and contractual aspects surrounding the software they create.

5.2 Users

Research software users are heterogeneous and can include individuals with different backgrounds and coming from disparate professions. However, the primary users of Research Software are usually researchers, research engineers, and scientists.

The diversity of users highlights the interdisciplinary nature of Research Software, which often spans various scientific fields and application domains. Adequate documentation, hands-on tutorials, and the steady availability of user support are crucial for ensuring that Research Software meets the users' needs and contributes to advancing knowledge in different fields.

Reproducibility of results

From the user's perspective, a crucial aspect is the reproducibility of the results of a published study in a scientific manuscript or project document. This has to do with the ability to access the software and data used to produce the results, i.e., the software artefact. We refer the reader to Section 2, where the definitions proposed by ACM, and followed in this report, are given.

Users must have access to the software source code to examine the algorithms used, verify their implementation, and understand how the software works in every aspect.To this end, the availability of comprehensive documentation and the long-term availability of the software artefact are essential. Indeed, users should have access to clear instructions on how to configure the software, install dependencies, compile the code, and reproduce specific experiments or analyses. Documentation of source code is certainly a plus to improve transparency and reproducibility of results.

The access to the data used in the study is essential for reproducing results through the Research Software. Users should have access to the same data used by the authors to run the software. Moreover, knowledge of the computational environment, including the operating system, software versions, dependencies, and hardware specifications used to obtain the results, is essential to facilitate the user to reproduce the results, this can be achieved by proper provenance metadata. Containers technology and virtual environments are more and more often used, when possible, to help recreate the exact computational environment.

However, users of Research Software should also recognize that exact reproducibility may not always be achievable due to some endogenous factors, such as data variability and different hardware-software platforms over time, or even due to exogenous factors related to the inherent randomness of the specific methodology used (e.g., Monte Carlo methods). Collaborative and transparent practices within the research community and between Research Software developers and the user community can help increase the culture of reproducibility by ensuring that results produced using Research Software are easily verifiable.

5.3 Service providers

In this section, an *external service provider* (or simply *service provider*) refers to a company or organization that offers services to assist individual researchers and research entities in developing

and releasing their Research Software. A service provider may offer either physical resources or software tools and infrastructures. Several service providers meet the needs of researchers and developers engaged in Research Software development. Examples include GitHub, Zenodo, and Slack, to mention a few. Many software developers use such external services to enhance collaboration, streamline software development, maintain and disseminate the software they developed.

However, although not every Research Software project needs to leverage services offered by external service providers, the possibility of relying on such services without worrying about the maintainability of the service itself is undoubtedly an excellent value for Research Software developers. For example, researchers often rely on cloud services, such as Amazon Web Services (AWS) and Google Cloud Platform (GCP), to access scalable computing infrastructures using a pay-as-you-go service model in which customers are charged on the basis of their actual usage or consumption of a service, rather than paying upfront a fixed fee. These computing infrastructures are usually complex to manage and maintain, requiring specialized expertise and a consistent budget. Additionally, they provide flexibility in handling different workloads and optimizing resource utilization. Therefore, service providers are crucial actors to enable the development and execution of the software and the fruition of the results produced by the Research Software.

External services that represent an added value for Research Software can materialize in various forms. We refer the reader to Section 4.1, for quality attributes recommendations related to services. The following are some particularly significant services.

- **Code hosting and version control**. Examples are GitHub and GitLab, which offer version control, collaboration features (e.g., issue tracking), and code sharing. These platforms help to maintain a transparent development process.
- **Continuous Integration and Deployment**. Examples are Travis CI and Jenkins. These services automate the testing and deployment of software, ensuring code quality and streamlining the release process.
- **Containerization Services**. Examples are Docker Hub, Podman, and Containerd which offer services that facilitate the packing, distribution, and sharing of Research Software along with its dependencies, thus enhancing the reproducibility, software portability, and scalability.
- **Cloud Computing Platforms**. Examples are AWS, Microsoft Azure, and GCP, which provide scalable infrastructure, enabling researchers to access computing resources on-demand.
- **Software and Data Repositories**. Examples include Zenodo, Dryad, and Figshare. They offer data and software archiving and sharing services, improving the accessibility of research artefacts. SoftwareHeritage, on the other hand, is an example of archiving and preserving the source code of software projects. The goal is to provide a reliable and persistent archive to ensure the availability of source code to future generations.
- **Collaboration Platforms**. Examples include Slack and Microsoft Teams, which offer communication and collaboration tools that improve team coordination, project management, and encourage real-time communication.
- **High-Performance Computing Centers**. Several national and regional HPC centers (e.g., CINECA and BSC) provide researchers with access to powerful computing resources,

enabling them to perform complex simulations and analyses that require significant computing power.

There are also code review, training, and consultancy services for improving software development and maintenance. For example, Code Review facilitates code review processes to help maintain code quality, identify issues, and ensure that contributions meet established standards. Software Carpentry provides workshops, training events, and educational resources to help researchers improve their software and data skills.

Given the large number of potentially valuable services for Research Software, an open problem is the fragmentation of these external services, i.e., the phenomenon whereby various software services, tools, or platforms are distributed and heterogeneous, which often results in the parcelization of services with poor interoperability and increased access costs. On the other hand, Research Software developers who invest in a single set of services from a specific vendor may become locked into a particular platform or ecosystem, thus limiting their flexibility and interoperability and introducing difficulties in migrating to alternative solutions. These phenomena can have important practical implications for Research Software developers that could be overcome by building open and interoperable multi-service infrastructures for Research Software.

6 Metadata for quality software

Software metadata has become a crucial asset when deploying and publishing software, as it describes the key aspects of a software package (e.g. the name, id, version, license, authors, etc.), aiding comparison against similar efforts and helping users with faceted search. Package managers such as PyPI,[7] Maven Central[8] or CRAN[9] provide ways of gathering metadata using their own representation.

With the increasing adoption of the FAIR principles [18], using metadata to describe research outcomes (including Research Software and code) is becoming a common practice. As with package managers, metadata make it easier for tool registries and aggregators to present Research Software with common descriptors that can be indexed and compared. Although metadata on their own does not guaranty quality, some metadata elements align to quality principles (e.g, documentation, licensing, version naming, etc.).

In Section 6.1 we review common metadata schemas. In Section 6.2, we discuss the relationship between FAIR and software code quality. Finally, in Section 6.3, we map some of the Software Quality Attributes with closely related FAIR principles for Research Software (FAIR4RS).

6.1 Metadata schemas for Software

A number of schemas, vocabularies and ontologies describe software at different levels of detail. The Description of a Project (DOAP) ontology[10] proposes metadata terms for capturing software projects, emphasizing how a project itself is managed (issues, bug tracking, wiki discussions, etc.). Other vocabularies such as OntoSoft [19] take on a scientist's perspective by capturing software metadata through a series of questions that researchers are familiar with. On a more abstract formalization level, the Core Software Ontology (CSO)[11] specifies terms for describing software and web services by extending the DOLCE upper ontology [20].

Other schemas have been specialized for a given domain. For example, the Software Ontology (SWO)[12] builds on the Open Biomedical Ontologies such as the Basic Formal Ontology [21] to describe software. These ontologies describe a thorough taxonomy of input and output types and formats, particular to the biomedical domain.

Finally, other community initiatives have opted for a lightweight approach. Codemeta [22] aims at creating a minimal metadata schema for Research Software and code. It is based on two types from `https://schema.org` [23], namely SoftwareSourceCode and softwareApplication. In addition, the Codemeta initiative has aligned metadata schemas from many software registries and repositories[13] making it easier to exchange records between them. Codemeta is one of the recommendations from the Scholarly infrastructures for Research Software [24], which calls for EU representatives to "help establish a stable, long-term governance" for the vocabulary. Two different initiatives have already proposed Codemeta extensions to capture the semantics of software tools. The first one is Bioschemas[14], a collaborative community project aiming at adding structured markup to Life Science websites, which provides a profile (i.e., a recommendation of

[7] `https://pypi.org/`
[8] `https://search.maven.org/`
[9] `https://cran.r-project.org/`
[10] `http://usefulinc.com/ns/doap`
[11] `http://km.aifb.kit.edu/sites/cos/`
[12] `http://purl.obolibrary.org/obo/swo.owl`
[13] `https://codemeta.github.io/crosswalk/`
[14] `https://bioschemas.org/`

usage created on top of a schema.org type) for describing ComputationalTools. The second one is the Software Description Ontology [25], which extends Codemeta with additional metadata fields to capture executable commands, configuration and limitations of a software component.

6.2 The FAIR principles for Research Software (FAIR4RS)

After the FAIR guiding principles for data were proposed [18], the scientific community discussed how to adapt them to different research artefacts. The FAIR Principles for Research Software Working Group[15] was created in September 2021 by the Research Data Alliance to adapt the FAIR principles to Research Software. As a result [26] summarizes the community discussion, defining the FAIR4RS principles as follows:

F: Software, and its associated metadata, is easy for both humans and machines to find.

- **F1** - Software is assigned a globally unique and persistent identifier.
  - **F1.1** - Components of the software representing levels of granularity are assigned distinct identifiers.
  - **F1.2** - Different versions of the software are assigned distinct identifiers.
- **F2** - Software is described with rich metadata.
- **F3** - Metadata clearly and explicitly include the identifier of the software they describe.
- **F4** - Metadata are FAIR, searchable and indexable.

A: Software, and its metadata, is retrievable via standardized protocols.

- **A1** - Software is retrievable by its identifier using a standardized communications protocol.
  - **A1.1** - The protocol is open, free, and universally implementable.
  - **A1.2** - The protocol allows for an authentication and authorization procedure, where necessary.
- **A2** - Metadata are accessible, even when the software is no longer available.

I: Software interoperates with other software by exchanging data and/or metadata, and/or through interaction via application programming interfaces (APIs), described through standards.

- **I1** - Software reads, writes and exchanges data in a way that meets domain-relevant community standards.
- **I2** - Software includes qualified references to other objects.

R: Software is both usable (can be executed) and reusable (can be understood, modified, built upon, or incorporated into other software).

- **R1** - Software is described with a plurality of accurate and relevant attributes.
  - **R1.1** - Software is given a clear and accessible license.
  - **R1.2** - Software is associated with detailed provenance.
- **R2** - Software includes qualified references to other software.

[15] https://www.rd-alliance.org/groups/fair-research-software-fair4rs-wg

- **R3** - Software meets domain-relevant community standards.

6.3 Mapping Quality Attributes to FAIR4RS

Many of the FAIR4RS principles described in Section 6.2 are tightly coupled with good practices for quality software and code. Table 16 aligns each of the FAIR4RS principles with the Software Quality Attributes shown in Appendix A. For each alignment we provide a brief explanation justifying the rationale of our decision.

FAIR4RS principle	Quality Attribute	Rationale
F1	EOSC-SWRelMan-11 EOSC-SWRelMan-13 EOSC-SWRelMan-25	A metadata file describing a software component should be appropriately identified and tied to its corresponding software project (e.g., on the source code repository). A software project package should be uploaded to a public registry (e.g., everytime there is a new release). The metadata file should contain the software identifier and its corresponding version Checksums must be used to identify each software binary file uniquely.
F1.1	EOSC-SWRelMan-05 EOSC-SWRelMan-33	Different branches should be used to address code changes with different scope. Each branch should be identified uniquely within a code repository. The documentation associated to a project must have a persistent identifier and linked to/from the metadata describing the software whenever needed (i.e., if the documentation resides outside the software, each one will have its own persistent identifier).
F1.2	EOSC-SWRelMan-02 EOSC-SWRelMan-04 EOSC-SWRelMan-06 EOSC-SWRelMan-09	A version control system should exist for the software project code, with a main identified branch where new features are merged. A strategy for transitioning between version releases should be in place. Changelogs and/or release notes should be used to keep trace of changes between versions/releases. Semantic versioning should be used to uniquely define each code release.
F2	EOSC-SWRelMan-11 EOSC-SWRelMan-26 EOSC-SWRelMan-29 EOSC-SWRelMan-30 EOSC-SWRelMan-31 EOSC-SWRelMan-32 EOSC-Qual-24	The software project should contain enough information for a reader to verify its objectives. A file including information about the authors (aka creators or contributors) should be provided (e.g., citation file) together with a metadata file describing the software project with links to/from the actual software (be aware that information about the project is not necessarily the same as for the software itself). Software project could benefit from a Software Management Plan. Documentation should be used to describe the Research Software component, using version control to track differences between software versions. A README file should be provided describing the project with metadata in a human-readable format (e.g., installation instructions, authors, citation, etc.).
F3	EOSC-SWRelMan-11	A metadata file should contain an identifier of the software project it describes.
F4	EOSC-SWRelMan-11	A metadata file should be provided to facilitate metadata propagation when releasing code (e.g. storing the code in Zenodo).
A1		
A1.1	EOSC-SWRelMan-01	The software project should be publicly available online.
A1.2	EOSC-SWTest-18	An authentication mechanism should be enabled for the target service. Metadata describing the software should include, when necessary, information on how to access the software (e.g., membership, payment, free of access) and complemented with a license (discussed later in this table).
A2	EOSC-SWRelMan-13	Metadata can be made accessible via registries. It is therefore a good idea to register the software in appropriate registries (e.g., some communities will have their own software registries or will suggest one of general purpose).
I1	EOSC-Qual-29 EOSC-SWTest-01	The software component should comply with internationally recognized standards. Code should follow standard best practices.
I2	EOSC-SWRelMan-11	Metadata should point to its documentation, tutorials, etc.
R1	EOSC-SWRelMan-11 EOSC-SWRelMan-29	A metadata file and documentation should describe the software project with accurate, relevant attributes.
R1.1	EOSC-SWRelMan-10 EOSC-SWRelMan-34 EOSC-SrvOps-01 EOSC-SrvOps-02	A license should be associated with the software project, especially if the project is open source. The documentation of a software project should also provide a license. In case a service is deployed, an "Acceptable Usage Policy" and "Terms of Use" policy should be clearly defined.
R1.2	EOSC-SWRelMan-29 EOSC-SWRelMan-30	Software documentation should motivate the rationale and organizations behind the development of a software project, as well as related initiatives that may have been derived from. Funding sources should also be acknowledged.
R2	EOSC-SWTest-25 EOSC-SWRelMan-12	Third party dependencies should be documented and checked for security flaws. A software project should be adequately packaged and installed, containing all the third party dependencies needed for doing so.
R3	EOSC-Qual-29 EOSC-SWRelMan-32	A software project should comply with internationally adopted standards. Project documentation should be adapted to the domain-relevant community. Metadata should also comply to community standards (e.g., Codemeta, Bioschemas, see more information in section 6.1).

Table 16: Mapping FAIR4RS to software quality attributes

7 Summary and conclusions

In this report we have tried to provide insights on how the quality of Research Software could be improved from several different points of view, as well as providing practical recommendations.

We avoided letting the professional views of the authors contributing to this document affect its objectivity. Indeed, each of us have had our own experience on Software development and/or operating software and services, which would not necessarily be representative or general enough for all users and situations.

To overcome this problem, we conducted a systematic and thorough survey of the existing literature according to the keywords, title, and field of the articles. The list was further processed to exclude non-suitable articles (for example those not actually discussing about quality aspects, among other criteria). Finally, we ended up with a set of relevant articles that were included in our study, which led to a classification into software quality characteristics and software quality attributes and metrics.

Given that the classification is motivated from what is found in significantly published peer-reviewed articles, this provided an unbiased list of characteristics, which we further classified into significant attributes and metrics. We summarised the list of attributed and metrics in Appendix A, where each entry contains our own proposed codename, its name in the source reference, the associated characteristics, its definition, its context of use, and a reference to the source article for tracking and proper citation.

The survey of the broad existing literature and the classification took a large time and effort, but we are convinced that it was the only proper way to proceed in order to describe the landscape of Research Software and, eventually, propose recommendations.

In the Introduction section of the document, we tried first to define what Research Software. Our study is specific for Research Software and therefore it was needed to give a definition. We realised that this attempt could probably end up in an endless discussion before reaching an agreement on a precise definition. Therefore, we took the definition for Research Software as from [2]; any software which is linked to a scientific publication, given that this is surely the most important of its characteristics.

Software systems can be, in general, organised in different ways, one of them being a stack of components and their dependencies. We reviewed the stack upon which software is built, from libraries to complete services and platforms.

To complete the landscape section, we included a discussion about what could be the expectations depending on the type of Research Software and depending on specific layers of the Software stack. From the first classification, one can clearly notice that the requirements for a team to build a complete services are very different from those of individual researchers who build their own scripts to produce results for a publication. In the second classification, we discussed the different contexts where Software could be considered, for example as a scientific structure or as a domain-specific tool.

The classification on characteristics, attributes, and metrics from the survey, along with the landscaping on Research Software, allowed us to propose practical recommendations.

Given the landscape of different users and scopes of Research Software, we wrote the recommendations at two different levels. The first group are general recommendations for all types of software including, of course, Research Software. In each group, we listed the corresponding quality attributes.

The second group of recommendations are related to what we have called *user stories*, and are recommendations depending on the role of the users. We have considered a user working in a team, a developer working on an open source project and a user developing a service or platform.

As a case of special interest, we added a section specifically about software in production stage, discussing the management of its release cycle and quality attributes related to services and platforms in production.

In order to facilitate understanding, we provide specific examples and use cases where the recommendations could be applied.

From the recommendations, it comes in a natural way the section about the perspectives or expectations of developers, users, and service providers with respect to Research Software. We profiled which kind of developers are typically working with Research Software, as well as its users. Last but not least, we included also the perspectives of service providers with specific examples of platforms and services.

Finally, we discuss about metadata of software and the FAIR principles and its relation with quality. We show examples of existing tools and initiatives to include metadata for software and discuss how this is important. With respect to the FAIR principles, we focus specifically on the FAIR4RS of the Research Software Working Group, and we map most of the FAIR4RS principles with the quality attributes we propose in this work.

The topology of all software is complex, and defining what should be the characteristics and attributes which define what we understand as its *quality* is challenging.

Indeed, not all quality criteria is equally applicable to different types of Research Software or in any context. The landscaping indeed shows this diversity, and we took that into account, when curating our proposed list of characteristics, attributes and metrics.

Our recommendations also take into account this diversity, completed with actual use case examples.

We hope the classification and recommendations given in this report will be useful to produce better Research Software to all actors involved.

A Quality Attributes

A.1 Source Code Metrics: EOSC-SCMet

EOSC-SCMet-01: Size of the software application: Maintainability.

- The software product's rebuild value is estimated from the number of lines of code. This value is calculated in man-years using the Programming Languages Table of the Software Productivity Research [27].

EOSC-SCMet-02: % of redundant code: Maintainability, Modifiability.

- A line of code is considered redundant if it is part of a code fragment (larger than 6 lines of code) that is repeated literally (modulo white-space) in at least one other location in the source code [27].

EOSC-SCMet-03: # Lines of code: Maintainability.

- Number of lines for the whole software project or components / modules / classes / functions / methods [28, 27].

EOSC-SCMet-04: % of assertions: Maintainability.

- Percentage of lines of source code containing assertions. Assertions are used as a means for demonstrating that the program is behaving as expected and as an indication of how thoroughly the source classes have been tested on a per class level [29].

EOSC-SCMet-05: Cyclomatic Complexity: Maintainability.

- Cyclomatic complexity (i.e., the number of linearly independent paths through a program's source code) of the whole software component or modules/components/classes/functions/methods. Cyclomatic complexity is created by calculating the number of different code paths in the flow of the program. A program that has complex control flow requires more tests to achieve good code coverage and is less maintainable [30, 28, 27, 29, 31].

EOSC-SCMet-06: Cyclomatic Complexity test/source ratio: Maintainability.

- Ratio between the sum of cyclomatic complexities of all tests and the source code [29].

EOSC-SCMet-07: Number of arguments: Maintainability.

- Number of arguments or parameters used in functions. Functions with many parameters may be a symptom of bad encapsulation [27, 31].

EOSC-SCMet-08: Number of function calls: Maintainability.

- Measures the inter-dependencies between unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration. Good software design dictates that types and methods should have high cohesion and low coupling. High coupling indicates a design that is difficult to reuse and maintain because of its many inter-dependencies on other types [28, 31].

EOSC-SCMet-09: Modularity: Maintainability, Functional suitability.

- Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. Also the number of modules [10, 28, 32, 31, 9].

EOSC-SCMet-10: Number of comments: Modifiability.

- Number of lines corresponding to comments for the whole software or per modules/components / classes / functions / methods [30, 28, 31].

EOSC-SCMet-11: Maintainability Index (MI): Maintainability.

- Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers. The Maintainability Index (MI) calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. Colour coded ratings can be used to quickly identify trouble spots in the code. A green rating is between 20 and 100 and indicates that the code has good maintainability. A yellow rating is between 10 and 19 and indicates that the code is moderately maintainable. A red rating is a rating between 0 and 9 [10, 28].

EOSC-SCMet-12: Internal cohesion: Maintainability.

- Cohesion describes how related the functions within a single module are. Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large [11, 28].

EOSC-SCMet-13: Test class to source class ratio: Reliability.

- Control measure to counter the confounding effect of class size. The coefficient is calculated by NOTC/NOSC (NOTC = number of test classes, NOSC = number of source classes) [29].

EOSC-SCMet-14: Coupling Between Objects (CBO) ratio: Maintainability.

- Ratio between the CBO in the tests and in the whole source code. The higher the inter-object class coupling, the more rigorous the testing should be [29].

EOSC-SCMet-15: Depth of inheritance Tree (DIT) ratio: Maintainability.

- Ratio between the DIT of the tests and the DIT of the source code. A higher DIT indicates desirable reuse but adds to the complexity of the code because a change or a failure in a super class propagates down the inheritance tree [29].

EOSC-SCMet-16: Weighted Methods per Class (WMC) ratio: Maintainability.

- Ratio between the WMC of the tests with respect to the WMC of the whole source code. This measure serves to compare the testing effort on a method basis. WMC is an indicator of fault-proneness [29].

EOSC-SCMet-17: Source lines of code (SLOC) ratio: Maintainability.

- SLOC of the whole project with respect to the minimum SLOC of its components [29].

A.2 Time and Performance Metrics: EOSC-TMet

EOSC-TMet-01: Effort required for changes: Reliability, Supportability.

- Time and resources dedicated to resolve an issue [28].

EOSC-TMet-02: # Resolved bugs: Supportability.

- Number of resolved bugs per period of time [28].

EOSC-TMet-03: # Open bugs: Supportability.

- Number of open bugs/issues per period of time [28].

EOSC-TMet-04: Defect rates: Maintainability.

- The number of outstanding defects in a product per period of time [33].

EOSC-TMet-05: Integrity: Integrity, Maintainability.

- Resource cost expended to solve problems caused by inconsistencies within the system. This may be measured in terms of staff time employed to fix problems and user time wasted. Also, degree to which a system, product or component prevents unauthorised access to, or modification of, computer programs or data [10, 34].

EOSC-TMet-06: Maintainability: Maintainability.

- Measured by the resources spent in terms of time and cost in keeping a system up and running over a period of time [34, 35].

EOSC-TMet-07: Adaptability: Reusability. Adaptability.

- Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments [10, 34, 35].

EOSC-TMet-08: # Registered users: Operability.

- The metric Number of registered users is to be collected [7].

EOSC-TMet-09: # Active users: Operability.

- The metric Number of active users over a given period of time may be collected [7].

EOSC-TMet-10: Amount of computing resources: Operability, Performance Efficiency.

- The metric amount of computing resources per user or group of users is collected. An example is CPU x hours, resource utilisation [10, 7].

EOSC-TMet-11: Amount of storage resources: Operability, Performance Efficiency.

- The metric amount of storage resources per user or group of users is collected. An example is GByte x hours. Resource utilisation and capacity [10, 7].

A.3 Qualitative Attributes: EOSC-Qual

EOSC-Qual-01: Complexity of diagrams: Maintainability. Reusability.

- Complexity of diagrams (UML) for the whole software or modules/components [28].

EOSC-Qual-02: Complexity of architecture: Maintainability. Reusability.

- Architecture showing modules and interactions [28, 36].

EOSC-Qual-03: Complexity of a use case: Maintainability. Reusability. Usability.

- Complexity of use case diagrams (UML) [28].

EOSC-Qual-04: Sustainable community: Supportability.

- An active community is behind the software product [32].

EOSC-Qual-05: User satisfaction: Attractiveness.

- Degree to which a user interface enables pleasing and satisfying interaction for the user [10, 36].

EOSC-Qual-06: Usability: Usability.

- Measured using user surveys. However, it may also be assessed in terms of calls upon support staff, e.g. number of requests for help or support staff time expended. Also, degree to which a product or system can be used by specified users to achieve specific goals with effectiveness, efficiency and satisfaction in a specified context of use [10, 36, 34, 35].

EOSC-Qual-07: Reliability: Reliability.

- The reliability of systems from the user's point of view is concerned with three things: (1) How often does it go wrong? (2) How long is it unavailable? (3) Is any information lost at recovery? Also, degree to which a system, product or component performs specific functions under specified conditions for a specified period of time [10, 34, 35].

EOSC-Qual-08: Timeliness: Supportability.

- Assessed in terms of the costs of non-delivery. These will possibly include staff time and lost sales. It may also be assessed in terms of the number of days departure from the date agreed with client [34].

EOSC-Qual-09: Cost-Benefit efficiency: Maintainability.

- Measured in simple financial terms. The costs of installing and maintaining the system are weighed against the assessment of business benefits [34].

EOSC-Qual-10: Accessibility: Technical accessibility.

- Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use [10, 35].

EOSC-Qual-11: Accountability: Performance, Resource utilisation.

- Degree to which the actions of an entity can be traced uniquely to the entity. Its usage can be measured; critical segments of code can be instrumented with probes to measure timing, whether specified branches are exercised, etc. [10, 35].

EOSC-Qual-12: Accuracy: Performance, Functional suitability.

- Code outputs are sufficiently precise to satisfy their intended use [35].

EOSC-Qual-13: Communicativeness: Ease of use.

- Metric which facilitates the specification of inputs and provides outputs whose form and content are easy to assimilate and useful [35, 8].

EOSC-Qual-14: Completeness: Supportability, Manageability.

- All software parts are present and each part is fully developed [10, 35].

EOSC-Qual-15: Conciseness: Supportability, Resource utilisation.

- Redundant information about the software code is not present [35].

EOSC-Qual-16: Consistency: Maintainability, Interoperability, Compatibility.

- Source code contains uniform notation, terminology and symbology within itself [35, 8].

EOSC-Qual-17: Legibility: Supportability, Maintainability.

- Software function is easily discerned by reading the code [35].

EOSC-Qual-18: Modifiability: Modifiability.

- Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality [10, 35].

EOSC-Qual-19: Robustness: Safety.

- Software can continue to perform despite some violation of the assumptions in its specification [35].

EOSC-Qual-20: Self-containedness: Supportability.

- Software performs all its explicit and implicit functions within itself [35].

EOSC-Qual-21: Self-descriptiveness: Supportability, Ease of use.

- Software contains enough information for a reader to determine or verify its objectives, assumptions, constraints, inputs, outputs, components, and revision status [35].

EOSC-Qual-22: Structuredness: Modifiability, Reusability.

- Software possesses a definite pattern of organisation of its interdependent parts [35].

EOSC-Qual-23: Understandability: Supportability.

- Software purpose is clear to the inspector [35].

EOSC-Qual-24: Intellectual Property: Supportability, Reusability.

- There are multiple statements embedded into the software product describing unrestricted rights and any conditions for use, including commercial and non-commercial use, and the recommended citation. The list of developers is embedded in the source code of the product, in the documentation, and in the expression of the software upon execution. The intellectual property rights statements are expressed in legal language, machine-readable code, and in concise statements in language that can be understood by laypersons, such as a pre-written, recognisable licence [9].

EOSC-Qual-25: Extensibility: Attractiveness, Modifiability.

- There is evidence that the software project has been extended externally by users outside of the original development group using existing documentation only. There is a clear approach for modifying and extending features across and in multiple scenarios, with specific documentation and features to allow the building of extensions which are used across a range of domains by multiple user groups. There may be a library available of user-generated content for extensions and user generated documentation on extension is also available [9].

EOSC-Qual-26: Standards compliance: Functional suitability.

- Compliance with open or internationally recognised standards for the software and software development process, is evident and documented, and verified through testing of all components. Ideally independent verification is documented through regular testing and certification from an independent group [9].

EOSC-Qual-27: Internationalisation and localisation: Usability.

- Demonstrable usability: Software has been tested with multiple pseudo or genuine translations [9].

A.4 DevOps-SW Release and Management Attributes: EOSC-SWRelMan

EOSC-SWRelMan-01: Open source : Supportability, Maintainability, Availability, Reusability.

- Following the open-source model, the source code being produced is open and publicly available to promote the adoption and augment the visibility of the software developments [6, 8].

EOSC-SWRelMan-02: Version Control System (VCS): Supportability, Maintainability.

- Source code uses a Version Control System (VCS). All software components delivered by the same project agree on a common VCS [6].

EOSC-SWRelMan-03: Source code hosting: Supportability, Maintainability.

- Source code produced within the scope of a broader development project resides in a common organisation of a version control repository hosting service [6].

EOSC-SWRelMan-04: Working state version: Maintainability.

- The main branch in the source code repository maintains a working state version of the software component. Main branch is protected to disallow force pushing, thus preventing untested and unreviewed source code from entering the production-ready version. New features are merged in the main branch whenever the agreed upon SQA criteria is fulfilled [6].

EOSC-SWRelMan-05: Changes branches: Maintainability.

- New changes in the source code are placed in individual branches. Branches follow a common nomenclature, usually by prefixing, to differentiate change types (e.g. feature, release, fix) [6].

EOSC-SWRelMan-06: Good patching practice: Maintainability.

- Secondary long-term branch that contains the changes for the next software release exists. Next release changes come from the individual branches. Once ready for release, changes in the secondary long-term branch are merged into the main branch and versioned. At that point in time, main and secondary branches are aligned. This step marks a production release [6, 8].

EOSC-SWRelMan-07: Support: Maintainability, Operability.

- Existence of an issue tracking system or helpdesk, facilitates structured software development. Leveraging issues to track down both new enhancements and defects (bugs, documentation typos). Applies as well to services to report operational and user issues [30, 9, 6, 7].

EOSC-SWRelMan-08: Code review: Maintainability.

- Code reviews are done in the agreed peer review tool within the project, with the following functionality: (a) Allows general and specific comments on the line or lines that need to be reviewed. (b) Shows the results of the required change-based test executions. (c) Allows to prevent merges of the candidate change whenever not all the required tests are successful.

Exceptions to this rule cover the third-party or upstream contributions which MAY use the existing mechanisms or tools for code review provided by the target software project. This exception is only allowed whenever the external revision life cycle does not interfere with the project deadlines [30, 32, 36, 6].

EOSC-SWRelMan-09: Semantic Versioning : Maintainability.

- Semantic Versioning specification is followed for tagging the production releases [6, 8].

EOSC-SWRelMan-10: Open-source licence: Supportability, Maintainability, Reusability.

- As open-source software, source code adheres to an open-source licence to be freely used, modified and distributed by others. Non-licensed software is exclusive copyright by default [6, 8].

EOSC-SWRelMan-11: Metadata: Supportability, Maintainability, Availability.

- A metadata file (such as Codemeta or Citation File Format) exists alongside the code, under its VCS. The metadata file is updated when needed, as is the case of a new version [6].

EOSC-SWRelMan-12: Packaging: Installability.

- Production-ready code is built as an artefact that can be installed on a system [9, 6, 8].

EOSC-SWRelMan-13: Register/publish artefact: Installability.

- The built artefact is uploaded and registered into a public repository of such artefacts [6].

EOSC-SWRelMan-14: Notification upon registration: Installability.

- Upon success of the package delivery process, a notification is sent to predefined parties such as the main developer or team [6].

EOSC-SWRelMan-15: Code deployment: Installability.

- Production-ready code is deployed as a workable system with the minimal user or system administrator interaction leveraging software configuration management (SCM) tools [6].

EOSC-SWRelMan-16: Software Configuration Management (SCM) as code: Installability.

- A software configuration management (SCM) module is treated as code. Version controlled, it resides in a different repository than the source code to facilitate the distribution [6].

EOSC-SWRelMan-17: SCM tool: Installability.

- All software components delivered by the same project agree on a common SCM tool. However, software products are not restricted to provide a unique solution for the automated deployment [6].

EOSC-SWRelMan-18: SCM code changes: Installability.

- Any change affecting the applications deployment or operation is subsequently reflected in the relevant SCM modules [6].

EOSC-SWRelMan-19: SCM official repositories: Installability.

- Official repositories provided by the manufacturer are used to host the SCM modules, thus augmenting the visibility and promote external collaboration [6].

EOSC-SWRelMan-20: Installability: Installability.

- Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment. Also, a production-ready SW or service is deployed as a workable system with the minimal user or system administrator interaction leveraging Infrastructure as Code templates [10, 7].

EOSC-SWRelMan-21: Preserve immutable infrastructures: Installability.

- Any future change to a deployed Service is done in the form of a new deployment, in order to preserve immutable infrastructures [7].

EOSC-SWRelMan-22: Infrastructure as Code (IaC) validation: Installability.

- IaC is validated by specific (unit) testing frameworks for every change being done [7].

EOSC-SWRelMan-23: Packaging of tarballs: Installability.

- Tarballs are not included in the distribution directory. Packaged tarballs are not extracted in the distribution directory [8].

EOSC-SWRelMan-24: Design for upgradability: Compatibility.

- Installation layout supports different versions and code releases [8].

EOSC-SWRelMan-25: Provide checksums: Maintainability.

- Code releases provide checksums with each binary (tarballs, RPMs, etc.) [8].

EOSC-SWRelMan-26: Documentation version controlled: Supportability.

- Documentation is version controlled. [7].

EOSC-SWRelMan-27: Documentation as code: Supportability, Maintainability, Reusability.

- Documentation is treated as code and resides in the same repository where the source code lies [6].

EOSC-SWRelMan-28: Documentation formats: Supportability, Maintainability, Reusability.

- Documentation uses plain text format using a markup language, such as Markdown or reStructuredText. All software components delivered by the same project agree on a common markup language [6, 8].

EOSC-SWRelMan-29: Documentation online: Supportability, Availability.

- Documentation is online available in a documentation repository. Documentation is rendered automatically [6, 7].

EOSC-SWRelMan-30: Documentation updates: Supportability, Maintainability, Reusability.

- Documentation is updated on new software or service versions involving any substantial or minimal change in the behaviour of the application [6, 7, 8].

EOSC-SWRelMan-31: Documentation updates if inaccurate/unclear: Supportability, Maintainability, Reusability.

- Documentation is updated whenever reported as inaccurate or unclear [6, 7].

EOSC-SWRelMan-32: Documentation production: Supportability, Maintainability, Reusability.

- Documentation is produced according to the target audience, varying according to the software component specification. The identified types of documentation and their content are README file(one-paragraph description of the application, a "Getting Started" step-by-step description on how to get a development environment running, automated test execution how-to, links to external documentation below, contributing code of conduct, versioning specification, author list and contacts, licence information and acknowledgements), Developer documentations (Private API documentation, structure and interfaces and build documentation), Deployment and Administration documentations (Installation and configuration guides, service reference card, FAQs and troubleshooting) and user documentations (Public API documentation and command-line reference) [32, 9, 6, 7, 8].

EOSC-SWRelMan-33: Documentation PID: Supportability.

- Documentation has a Persistent Identifier (PID) [7].

EOSC-SWRelMan-34: Documentation licence: Supportability.

- Documentation has a non-software licence [7].

A.5 DevOps - Testing Attributes: EOSC-SWTest

EOSC-SWTest-01: Code style: Maintainability, Testability.

- Each individual software complies with a de-facto code style standard for all the programming languages used in the codebase. Multiple standard style compliance is possible [6, 8].

EOSC-SWTest-02: Unit tests: Maintainability, Testability.

- Minimum acceptable code coverage threshold is 70%. Unit testing coverage is higher for those sections of the code identified as critical by the developers, such as units part of a security module. Unit testing coverage may be lower for external libraries or pieces of code not maintained within the product's code base [32, 29, 35, 9, 6, 8].

EOSC-SWTest-03: Test doubles: Functional suitability, Testability.

- When working on automated testing, Test Doubles (i.e., objects or procedures that look and behave like their release-intended counterparts, but are actually simplified versions that reduce the complexity and facilitate testing) are used to mimic a simplistic behaviour of objects and procedures [6, 7].

EOSC-SWTest-04: Test-Driven Development (TDD): Functional suitability, Maintainability, Testability.

- Software requirements are converted to test cases, and these test cases are checked automatically. Also, degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met [10, 33, 36, 6].

EOSC-SWTest-05: API testing: Functional suitability, Testability.

- API testing MUST cover the validation of the features outlined in the specification (aka contract testing) [7].

EOSC-SWTest-06: Integration testing: Functional suitability, Testability, Interoperability.

- Whenever a new functionality is involved, integration testing guarantees the operation of any previously-working interaction with external services [10, 7].

EOSC-SWTest-07: Functional testing: Functional suitability, Testability.

- Functional testing covers the full scope -e.g. positive, negative, edge cases- for the set of functionality that the software or service claims to provide [10, 7].

EOSC-SWTest-08: Performance testing: Functional suitability, Testability.

- Performance testing is carried out to check the Software Application performance under varying loads [10, 7].

EOSC-SWTest-09: Stress testing: Functional suitability, Testability.

- Stress testing is carried out to check the Service to determine the behavioural limits under sudden increased load [7].

EOSC-SWTest-10: Scalability testing: Functional suitability, Testability.

- Scalability testing is carried out to check the Service ability to scale up and/or scale out when its load reaches the limits [7].

EOSC-SWTest-11: Elasticity testing: Functional suitability, Testability.

- Elasticity testing is carried out to check the Service ability to scale out or scale in, depending on its demand or workload [7].

EOSC-SWTest-12: Open Web Application Security Project (OWASP): Security.

- Application is compliant with Open Web Application Security Project (OWASP) secure coding guidelines [6].

EOSC-SWTest-13: Static Application Security Testing (SAST): Security.

- Source code uses automated linter tools to perform static application security testing (SAST) that flag common suspicious constructs that may cause a bug or lead to a security risk (e.g. inconsistent data structure sizes or unused resources) [6].

EOSC-SWTest-14: Security code reviews: Security.

- Security code reviews for certain vulnerabilities is done as part of the identification of potential security flaws in the code. Inputs come from automated linters and manual penetration testing results [34, 9, 6].

EOSC-SWTest-15: No world-writable files or directories: Security.

- World-writable files or directories are not present in the product's configuration or logging locations [6].

EOSC-SWTest-16: Public endpoints and APIs encrypted: Security.

- The Service public endpoints and APIs are secured with data encryption [7].

EOSC-SWTest-17: Strong ciphers: Security.

- The Service uses strong ciphers for data encryption [7].

EOSC-SWTest-18: Authentication and Authorisation: Security, Technical accessibility.

- The Service has an authentication and authorisation mechanism. The Service validates the credentials and signatures [9, 7].

EOSC-SWTest-19: API security assessment: Security.

- API testing includes the assessment of the security-related criteria [7].

EOSC-SWTest-20: Service compliance with data regulations (GDPR): Security.

- The Service handles personal data in compliance with the applicable regulations, such as the General Data Protection Regulation (GDPR) within the European boundaries [7].

EOSC-SWTest-21: Dynamic Application Security Testing (DAST): Security.

- DAST checks are executed, through the use of ad-hoc tools, directly to an operational Service in order to uncover runtime security vulnerabilities and any other environment-related issues (e.g. SQL injection, cross-site scripting or DDOS). The latest release of OWASP's Web Security Testing Guide and the NIST's Technical Guide to Information Security Testing and Assessment are considered for carrying out comprehensive Service security testing [7].

EOSC-SWTest-22: Interactive Application Security Testing (IAST): Security.

- Interactive Application Security Testing (IAST), analyses code for security vulnerabilities while the app is run by an automated test. IAST is performed to a service in an operating state [7].

EOSC-SWTest-23: Security penetration testing: Security.

- Penetration testing (manual or automated) is part of the application security verification effort [7].

EOSC-SWTest-24: Security assessment: Security.

- The security assessment of the target system configuration is particularly important to reduce the risk of security attacks. The benchmarks delivered by the Centre for Internet Security (CIS) and the NIST's Security Assurance Requirements for Linux Application Container Deployments are considered for this task. Also, the degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorisation [10, 7].

EOSC-SWTest-25: Security as Code (SaC) Testing: Security.

- IaC testing covers the security auditing of the IaC templates (SaC) in order to assure the deployment of secured Services. For all the third-party dependencies used in the IaC templates (including all kinds of software artefacts, such as Linux packages or container-based images) [7].

A.6 Service Operability Attributes: EOSC-SrvOps

EOSC-SrvOps-01: Acceptable Usage Policy (AUP): Supportability.

- An Acceptable Usage Policy (AUP) is declared. AUP Is a set of rules applied by the owner, creator or administrator of a network, Service or system, that restrict the ways in which the network, Service or system may be used and sets guidelines as to how it should be used. The AUP can also be referred to as: Acceptable Use Policy or Fair Use Policy [7].

EOSC-SrvOps-02: Access Policy and Terms of Use: Supportability.

- An Access Policy or Terms of Use (APTU) is declared. APTU represents a binding legal contract between the users (and/or customers), and the Provider of the Service. The Access Policy mandates the users (and/or customers) access to and the use of the Provider's Service [7].

EOSC-SrvOps-03: Privacy policy: Supportability.

- A Privacy Policy is declared, with a data privacy statement informing the users (and/or customers), about which personal data is collected and processed when they use and interact with the Service. It states which rights the users (and/or customers) have regarding the processing of their data [7].

EOSC-SrvOps-04: Operational Level Agreement (OLA): Supportability.

- The Service includes an Operational Level Agreement (OLA) with the infrastructure where it is integrated [7].

EOSC-SrvOps-05: Service Level Agreement (SLA): Supportability.

- The Service includes Service Level Agreement (SLA) with user communities [7].

EOSC-SrvOps-06: Monitoring service public endpoints: Availability, Reliability.

- The Service public endpoints are monitored, such as probes measuring the http or https response time [7].

EOSC-SrvOps-07: Monitoring service public APIs: Availability, Reliability.

- The Service public APIs are monitored through the use of functional tests [7].

EOSC-SrvOps-08: Monitoring service Web Interface: Availability, Reliability.

- The Service Web interface is monitored, this can be accomplished through the use of automated and periodic functional tests [7].

EOSC-SrvOps-09: Monitoring security public endpoints and APIs: Availability, Reliability.

- The Service is monitored for public endpoints and APIs secured and strong ciphers for encryption [7].

EOSC-SrvOps-10: Monitoring security DAST: Availability, Reliability.

- The Service is monitored with DAST security checks [7].

EOSC-SrvOps-11: Infrastructure monitoring: Availability, Reliability.

- The Service is monitored for infrastructure-related criteria [7].

EOSC-SrvOps-12: Monitoring with Unit tests: Availability, Reliability.

- IaC (unit) tests are reused as monitoring tests, thus avoiding duplication [7].

References

[1] Mario David et al. *EOSC-SYNERGY. EU DELIVERABLE: D3.1 Software Maturity baseline.* EOSC-SYNERGY receives funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 857647., 2020. DOI: 10.20350/digitalCSIC/12607.

[2] Morane Gruenpeter et al. *Defining Research Software: a controversial discussion.* Zenodo, Sept. 13, 2021. DOI: 10.5281/zenodo.5504016. URL: https://zenodo.org/record/5504016 (visited on 01/28/2022).

[3] Guy Courbebaisse et al. *Research Software Lifecycle.* Sept. 2023.

[4] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[5] B. Kitchenham and S. Charters. *Guidelines for performing systematic literature reviews in software engineering.* 2007.

[6] Pablo Orviz et al. "A set of common software quality assurance baseline criteria for research projects". In: (2017). In collab. with Digital.CSIC and Digital.CSIC. DOI: 10.20350/DIGITALCSIC/12543. URL: https://digital.csic.es/handle/10261/160086 (visited on 02/10/2022).

[7] Pablo Orviz Fernández et al. "EOSC-synergy: A set of Common Service Quality Assurance Baseline Criteria for Research Projects". In: (June 15, 2020). In collab. with Digital.CSIC and Digital.CSIC. DOI: 10.20350/DIGITALCSIC/12533. URL: https://digital.csic.es/handle/10261/214441 (visited on 02/10/2022).

[8] Eric Steven Raymond. *Software Release Practice HOWTO.* Jan. 14, 2013. URL: https://tldp.org/HOWTO/Software-Release-Practice-HOWTO/.

[9] John Shepherdson. "CESSDA Software Maturity Levels". In: (Mar. 29, 2019). DOI: 10.5281/zenodo.2614050. URL: https://zenodo.org/record/2614050 (visited on 02/10/2022).

[10] ISO Central Secretary. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models: ISO/IEC 25010:2011.* ISO. 2017. URL: https://www.iso.org/standard/35733.html (visited on 06/24/2022).

[11] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary". In: *ISO/IEC/IEEE 24765:2017(E)* (2017), pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712.

[12] Microsoft. *Design Fundamentals.* 2010. URL: https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ee658094(v=pandp.10) (visited on 2010).

[13] Gartner. *Gartner Glossary: Scalability.* 2023. URL: https://www.gartner.com/en/information-technology/glossary/scalability (visited on 2021).

[14] André B. Bondi. "Characteristics of Scalability and Their Impact on Performance". In: *Proceedings of the 2nd International Workshop on Software and Performance.* WOSP '00. Ottawa, Ontario, Canada: Association for Computing Machinery, 2000, pp. 195–203. ISBN: 158113195X. DOI: 10.1145/350391.350432. URL: https://doi.org/10.1145/350391.350432.

[15] Konrad Hinsen. "Dealing With Software Collapse". In: *Computing in Science and Engineering* 21.3 (2019), pp. 104–108. DOI: 10.1109/MCSE.2019.2900945.

[16] Daniel Le Berre et al. *Higher Education and Research Forges in France - Definition, uses, limitations encountered and needs analysis.* Tech. rep. Comité pour la science ouverte, May 2023. DOI: 10.52949/37. URL: https://hal-lara.archives-ouvertes.fr/hal-04208924.

[17] Jeremy Cohen et al. "The Four Pillars of Research Software Engineering". In: *IEEE Software* 38.1 (2021), pp. 97–105. DOI: 10.1109/MS.2020.2973362.

[18] Mark D. Wilkinson et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data* 3.1 (Mar. 15, 2016), p. 160018. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18. URL: https://www.nature.com/articles/sdata201618 (visited on 06/02/2023).

[19] Yolanda Gil et al. "OntoSoft: A distributed semantic registry for scientific software". In: *e-Science (e-Science), 2016 IEEE 12th International Conference on*. IEEE. 2016, pp. 331–336. DOI: 10.1109/eScience.2016.7870916. URL: https://dgarijo.com/papers/ontoSoft2016.pdf.

[20] Aldo Gangemi et al. "Sweetening ontologies with DOLCE". In: *International conference on knowledge engineering and knowledge management*. Springer. 2002, pp. 166–181.

[21] Andrew D Spear, Werner Ceusters, and Barry Smith. "Functions in basic formal ontology". In: *Applied Ontology* 11.2 (2016), pp. 103–128.

[22] *CodeMeta Project Crosswalk*. https://codemeta.github.io/. 2017. URL: https://codemeta.github.io/.

[23] R. V. Guha, Dan Brickley, and Steve Macbeth. "Schema.Org: Evolution of Structured Data on the Web". In: *Commun. ACM* 59.2 (2016), pp. 44–51. ISSN: 0001-0782. DOI: 10.1145/2844544. URL: https://doi.org/10.1145/2844544.

[24] European Commission, Directorate-General for Research, and Innovation. *Scholarly Infrastructures for Research Software - Report from the EOSC Executive Board Working Group (WG) Architecture Task Force (TF) SIRS*. Publications Office, 2020. DOI: https://doi.org/10.2777/28598.

[25] Daniel Garijo et al. "OKG-Soft: An open knowledge graph with machine readable scientific software metadata". In: *2019 15th International Conference on eScience (eScience)*. IEEE. 2019, pp. 349–358.

[26] Neil P. Chue Hong et al. *FAIR Principles for Research Software (FAIR4RS Principles)*. Version 1.0. May 2022. DOI: 10.15497/RDA00068. URL: https://doi.org/10.15497/RDA00068.

[27] Robert Baggen et al. "Standardized code quality benchmarking for improving software maintainability". In: *Software Quality Journal* 20.2 (June 2012), pp. 287–307. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/s11219-011-9144-9. URL: http://link.springer.com/10.1007/s11219-011-9144-9 (visited on 06/24/2022).

[28] Sonia Montagud, Silvia Abrahão, and Emilio Insfran. "A systematic review of quality attributes and measures for software product lines". In: *Software Quality Journal* 20.3 (Sept. 2012), pp. 425–486. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/s11219-011-9146-7. URL: http://link.springer.com/10.1007/s11219-011-9146-7 (visited on 06/24/2022).

[29] Nachiappan Nagappan et al. "Early estimation of software quality using in-process testing metrics: a controlled case study". In: *Proceedings of the third workshop on Software quality - 3-WoSQ*. the third workshop. St. Louis, Missouri: ACM Press, 2005, p. 1. ISBN: 978-1-59593-122-1. DOI: 10.1145/1083292.1083304. URL: http://dl.acm.org/citation.cfm?doid=1083292.1083304 (visited on 06/24/2022).

[30] Kamonphop Srisopha and Reem Alfayez. "Software quality through the eyes of the end-user and static analysis tools: a study on Android OSS applications". In: *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, May 28, 2018, pp. 1–4. ISBN: 978-1-4503-5737-1. DOI: 10.1145/3194095.3194096. URL: https://dl.acm.org/doi/10.1145/3194095.3194096 (visited on 06/24/2022).

[31] H. Ogasawara, A. Yamada, and M. Kojo. "Experiences of software quality management using metrics through the life-cycle". In: *Proceedings of IEEE 18th International Conference on Software Engineering*. Berlin, Germany: IEEE Comput. Soc. Press, 1996, pp. 179–188. ISBN: 978-0-8186-7247-7. DOI: 10.1109/ICSE.1996.493414. URL: http://ieeexplore.ieee.org/document/493414/ (visited on 07/28/2023).

[32] Mark Aberdour. "Achieving Quality in Open-Source Software". In: *IEEE Software* 24.1 (Jan. 2007), pp. 58–64. ISSN: 0740-7459. DOI: 10.1109/MS.2007.2. URL: http://ieeexplore.ieee.org/document/4052554/ (visited on 06/24/2022).

[33] Lisa Crispin. "Driving Software Quality: How Test-Driven Development Impacts Software Quality". In: *IEEE Software* 23.6 (Nov. 2006), pp. 70–71. ISSN: 0740-7459. DOI: 10.1109/MS.2006.157. URL: http://ieeexplore.ieee.org/document/4012627/ (visited on 06/24/2022).

[34] Alan Gillies. "Modelling software quality in the commercial environment". In: *Software Quality Journal* 1.3 (Sept. 1992), pp. 175–191. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/BF01720924. URL: http://link.springer.com/10.1007/BF01720924 (visited on 06/24/2022).

[35] B.W. Boehm, J.R. Brown, and M. Lipow. "Quantitative evaluation of software quality". In: *Proceedings - 2nd International Conference on Software Engineering, ICSE 1976*. San Francisco; United States, Oct. 13, 1976, pp. 592–605.

[36] Wolfgang Zuser, Stefan Heil, and Thomas Grechenig. "Software quality development and assurance in RUP, MSF and XP: a comparative study". In: *Proceedings of the third workshop on Software quality - 3-WoSQ*. the third workshop. St. Louis, Missouri: ACM Press, 2005, p. 1. ISBN: 978-1-59593-122-1. DOI: 10.1145/1083292.1083300. URL: http://dl.acm.org/citation.cfm?doid=1083292.1083300 (visited on 06/24/2022).