Software Quality Assurance as a Service: Encompassing the Quality Assessment of Software and Services

Samuel Bernardo[a], Pablo Orviz[b], Mario David[a], Jorge Gomes[a], David Arce[c], Diana Naranjo[c], Ignacio Blanquer[c], Isabel Campos[b], Germán Moltó[c], Joao Pina[a]

[a]*Laboratory of Instrumentation and Particles - LIP, Av. Prof. Gama Pinto, 2, Lisbon, 1649-003, Portugal*
[b]*Institute of Physics of Cantabria - CSIC - UC, Av. de los Castros s/n, Santander, 39005, Spain*
[c]*Instituto de Instrumentación para Imagen Molecular (I3M). CSIC – Universitat Politècnica de València. Camino de Vera S/N, Valencia, 46022, Spain*

Abstract

This paper introduces the Software Quality Assurance as a Service (SQAaaS) concept and it describes an open-source implementation of a comprehensive platform that supports the automated assessment of specific quality metrics for software and services, defined as a set of baseline requirements. The platform is openly accessible, focuses on research software and open science, and promotes best practices by awarding standards-based digital badges to software and services. It provides an easy-to-use web-based graphical user interface which facilitates the interaction with server-side components in charge of automatically creating CI/CD (Continuous Integration / Continuous Delivery) pipelines for automated testing of the baseline criteria. The service is in production and has performed over 2800 assessments, awarding more than 125 digital badges across several scientific disciplines.

Keywords:
Software quality Software Sustainability, Digital badges, Open science
PACS: 0000, 1111
2000 MSC: 0000, 1111

1. Introduction

Software quality understood as reliability, sustainability, - and reusability, is a fundamental aspect both for industry and research [1]. In particular, quality assurance is an essential trait of software development. It allows users and managers to have higher trust that the software and related services will work as supposed, provide the expected results and meet their requirements. Furthermore, it also contributes to sustainability and facilitates collaboration between software developers.

Our focus is on research software development, i.e. broadly speaking, software developed with the purpose to be used in scientific environments: from simulation to analysis software, as well as middleware software used to deploy services supporting advanced scientific computing.

As stated in the work of Axelsson et al. [2], the quality of a software product stands for the fulfilment of the needs and requirements defined by the stakeholders. In this sense, quality in research software is not the same as quality in industry-oriented software, the latter being much more structured around "processes", while the first crucially relies on creative evolution, and less on standard development processes. In the world of research, discussions on the long-term maintenance effort for the software are often pushed aside for well-known reasons such as: lack of time and resources, development done by PhD students and postdocs moving to other jobs, availability of key people, or insufficient recognition in the career path as in the case of early career researchers. However, research software still needs to be "fit for purpose".

There is a broad consensus on the necessity to encourage the general adoption of best practices around research software development. We want to stress that the current problems of software sustainability in research, far from wearing off, will become more acute due to the increasing dependency of Science on software. Therefore, the development processes in Science must become more sustainable. One important premise of our work is that there are key points behind Industrial Software Quality which should be transmitted to the development of research software, notably the concepts of "Maintenance" and "Return on Investment". To this aim, quality assurance represents the activities carried out towards achieving and validating the desired level of quality [3].

In this paper, we describe the architecture and practical implementation of a platform to facilitate software and services quality assessment, which we have named Software Quality Assurance as a Service (SQAaaS)[1]. We follow a fully structured approach encompassing the quality of software and services, which constitutes one of the main novelties of the developments described here. As it will be noted, our decision-making and architectural choices are strongly influenced towards supporting the Open Science paradigm [4]. Our developments are, however applicable to general software development and service testing and deployment processes in science and engineering and is based on three pillars:

[1]SQAaaS - `https://sqaaas.eosc-synergy.eu`

(a) Identification and adoption of best practices for software development, and related services operation establishing quality criteria that are both open and technology agnostic. These criteria are based on widely adopted best practices that have been collected and successfully applied in many environments both in academia and industry.

(b) Development of libraries, tools and a Quality Assurance platform following DevOps principles and exploiting a Continuous Integration and Continuous Delivery (CI/CD) process to enable automated verification of the quality criteria identified in (a). These developments are usable both programatically through APIs for advanced usage and more easily through a web interface.

(c) The level of adherence of the software, services and data repositories towards best practices identified by (b) will be rewarded through verifiable, shareable and portable quality badges following the Open Badges specification [5]. These badges highlight the achievements to both promote adherence to quality and make the quality achievements explicitly visible to the users, therefore raising awareness.

After the introduction, the remainder of the paper is structured as follows. First, section 2 summarizes the state of the art and motivation as well as the rationale behind the software and services maturity baselines. Then, section 3 describes the strategy of using digital badges to recognize software and services developed according to quality criteria. Next, section 4 describes the architecture of the Software Quality Assurance as a Service (SQAaaS) framework developed, whereas its high-level functionalities are described in section 5. After that, section 6 discusses the technical decisions and development of the core components of the platform. Later, section 7 describes the user interface developed to facilitate its usage. Finally, section 8 provides business logic examples while section 10 summarizes the main achievements and concludes with the current status and future work.

2. Quality Assessment in Software

Quality assessment is an important trait for software and for services. It allows users and managers to have higher trust that, during its use and operation, the software and related services will work as supposed, give the expected results and meet their requirements. Furthermore, it also contributes to the maintainability, stability and sustainability of the software and services. Finally, it facilitates the collaboration between software developers and promotes good software development practices.

2.1. State of the Art

The most relevant quality model [6] is the standard defined in the "ISO/IEC 25010:2011(en) Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models" [7]. It replaces the ISO/IEC 9126-1:2001. The ISO/IEC 25010:2011(en) defines the following two models:

- A **Quality in Use model** composed of five characteristics (some of which are further subdivided into sub-characteristics) that relate to the outcome of interaction when a product is used in a particular context of use: 1) Effectiveness, 2) Efficiency, 3) Satisfaction, 4) Freedom from risk and 5) Context coverage.

- A **Product Quality model** composed of eight characteristics (which are further subdivided into sub-characteristics) that relate to static properties of software and dynamic properties of the computer system: 1) Functional suitability, 2) Performance efficiency, 3) Compatibility, 4) Usability, 5) Reliability, 6) Security, 7) Maintainability and 8) Portability.

The Quality in Use model seeks to quantify the "usability" (effectiveness, efficiency and satisfaction) of the application, when specific users attempt to meet their specified goals. The quantification of the characteristics of this model, are user based. The Product Quality model characteristics are of more interest to the developers of the product (or Software component). The implementation and documentation of plans to quantify and verify each of the characteristics, are the responsibility of the developers or supplier of the product.

On the one hand both the Quality in Use and the Product Quality models only tackle external quality and require the Software in execution within a given environment. On the other hand, only some of the Product Quality model characteristics are appropriate for automation.

The basis for the standard was connected to Commercial Off The Shelf (COTS) software. The Open Source Software (OSS) has characteristics that are not present in COTS, such as public access to the source code and participation of community members (both the development and user side) [8]. Several such models are reviewed in [9]. Some have their origin in the ISO/IEC 9126-1:2001, some are hierarchical and some are based on a Maturity model. The metrics corresponding to the assessment of the quality, are in general based on a given algorithm implemented in tools that automate the assessment of the quality.

The DevOps approach links development and operations for software components, through the use of a Continuous Integration and Continuous Delivery pipeline. The CAMS model stands for Culture, Automation, Measurement and Sharing, which are named as the four-fundamental dimensions to enable DevOps [10]. It does not have a single standard but takes the best practices from several standards. The authors of [11] propose a Quality model based on DevOps, while it reviews some of the other Quality models. The metrics are based on several sources and are posed as questions. These questions/metrics focus on the number of features delivered, the time a feature needs to be delivered or the number of releases to deliver these features. They also map most of the metrics with the Product Quality model of the ISO/IEC 25010:2011 [7].

In our work, in order to encompass the requirements of software development, we follow the DevOps approach that links development and operations through the use of Continuous Integration and Continuous Delivery pipelines, building on best practices from several standards.

We have defined two Quality Models, one for software [12] and one for services [13], establishing the pragmatic minimum set of principles to attain and assess quality. The Quality Models define the attributes for the Software Quality Assurance as a Service (SQAaaS) platform, aiming to make quality verification and assessment easily accessible to software developers and service integrators.

2.2. Software Quality Assurance baseline

The software quality baseline is focused on software development. It is based on first-hand experiences and widely adopted best practices in industry and research, which have been collected and successfully applied in several research projects [14, 15]. The definition of quality baselines benefits from the input of software developers and, therefore is a process in continuous evolution. In the context of this work, we gathered additional criteria from research software developers in the areas of Environment, Earth Observation, Astrophysics, and Life Sciences [16] and feedback from the EOSC Task Force on Research Software Infrastructures [17].

The set of minimum software quality criteria follows the spirit of enhancing visibility, accessibility and distribution of the source code through the alignment to the Open Source Definition [18], in particular promoting code style standards to deliver good quality source code emphasizing its readability and re-usability.

In our approach, DevOps principles improve the quality and reliability of software by covering different testing methods at the development and pre-production stages. Following this, we propose a change-based driven scenario where all new updates in the source code are continuously validated by the automated execution of the relevant tests. The minimum set of quality attributes also encourages secure coding practices and Static Application Security Testing (SAST) at the development phase while providing recommendations on external security assessment.

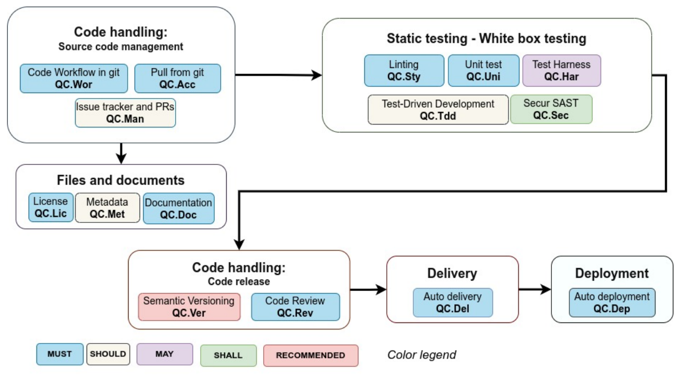


Figure 1: Software Quality Assurance workflow.

This section provides a high-level summary of the software quality criteria baseline. A full description including sub-criteria can be found in “A set of Common Software Quality Assurance Baseline Criteria for Research Projects”[2] [19]. The logical workflow of the Software Quality Assurance process is depicted in Figure 1. The keywords *MUST, SHOULD, MAY, SHALL and RECOMMENDED* are to be interpreted as described in RFC 2119 [20].

There are five main blocks involved in the workflow for the quality model.

- **Code handling - Source code management**, includes the criteria for code accessibility, workflow and management. In the quality model, software must be open and publicly available to promote adoption and augment the visibility of the developments. As such, it must adhere to an open-source license. An issue tracking system, including a system to pull or merge requests are also considered best practice to review and discuss the proposed changes. Each individual software product must also comply with community-driven or de-facto code style standards for the programming languages being used. Documentation must exist and be accessible. Metadata for the software component provides a way to achieve full identification, thus making software citable [21]. A metadata file should exist alongside the code, included in the version control system. The specifically required files (license, metadata, and documentation) are included in the block of files and documents. Documentation and License are a "must" to have a basic quality level.

- **Static testing - White box testing**, includes code style checking and testing. Unit testing evaluates all the possible flows of the code so that its fit for use can be automatically assessed. Test harness complements the implementation of Unit testing when stubs and mocks are necessary to isolate the unit being tested. Test driven development relies on implementation of tests based on software requirements.

- **Code handling - Code release**, includes checking of semantic versioning and code review. Code review implies the informal, non-automated, peer review of any change in the source code. It appears as the last step in the change management pipeline and must be done in a peer review tool previously agreed by the product team.

- **Delivery**, is the packaging of the Software into an artifact and publishing in a public repository for such artifacts.

- **Deployment**, which fetches the artifact from its public repository and deploys it into a working state.

2.3. Services Quality Assurance baseline

The meaning of service can be regarded from different perspectives. From an IT Service Management (ITSM) [22] standpoint a service is devised as a means to provide value to the customer. In the ITSM model a service is an intangible asset that

[2]see also `https://github.com/indigo-dc/sqa-baseline`

also includes additional activities such as customer engagement and support. Consequently, is a heavy process that might not be appropriately applicable for all types of services. The DevOps model, on the other hand, narrows down the scope to meet the user expectations by acting exclusively on the quality features of the service, which is seen as an aggregate of software components in operation.

In this work we propose harnessing the capabilities of the quality factors in the underlying software to lay out the principles for attaining quality in the services. According to this view, service quality is the foundation for shaping user-centric, reliable and fit-for-purpose services. The quality baseline for services is therefore designed to favour a pragmatic and systematic approach that emphasizes the programmatic assessment of the quality conventions. To this end, the Service Quality criteria builds on the preceding Software Quality Assurance baseline, to outline the good practices that seek the usability and reliability of services and meet the user expectations regarding functional requirements.

By leveraging a DevOps approach, the Service Quality baseline complements the Software Quality Model, with the existing approaches to assess and assure the quality and maturity of services, i.e. Technology Readiness Levels (TRLs) and general Service Management System (SMS). It builds trust on the users by strengthening the reliability and stability of the services, with a focus on the underlying software, thus ensuring a proper realization of the verification and validation processes. The functional suitability of the service is ensured by promoting testing techniques that check the compliance of the user requirements. Examples of services, as conceived in this quality baseline can be a Web service [3], a Web Application [4] or a Platform [5] created by the aggregation of multiple smaller services.

The logical workflow of the Service Quality Assurance process is depicted in Figure 2. In the following we summarize the quality conventions and best practices that apply to production service deployment. A more detailed description including sub-criteria can be found in the work by Orviz et al. [13] [6].

The Service Quality Model is divided into two main categories; Automation criteria and Operational criteria. Automation criteria are fit for execution in a pipeline, while Operational criteria are to be verified on a service in production.

The automated deployment of services implies the use of code to install and configure them in the target environment or infrastructure. Infrastructure as Code (IaC) [23] templates allow operations teams to treat service provisioning and deployment similarly to software code management. Consequently, IaC enables the paradigm of immutable infrastructure deployment and maintenance, where services are never updated, but deprovisioned and redeployed. An immutable infrastructure simplifies maintenance and enhances repeatability and reliability. Therefore it's a feature that services should have.

[3] `https://techterms.com/definition/web_service`
[4] `https://techterms.com/definition/web_application`
[5] `https://csrc.nist.gov/glossary/term/Service_Composition`
[6] See also `https://github.com/EOSC-synergy/service-qa-baseline`

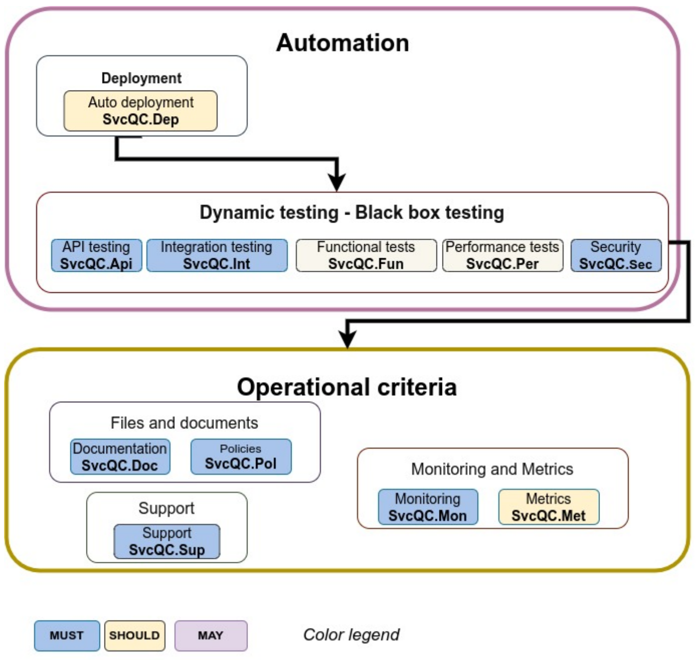


Figure 2: Service Quality Assurance workflow

The next step of the workflow is the Black box testing, where API testing, security and integration testing must be performed. API testing assumes the presence of an open API specification to make possible the validation. The Integration testing refers to the evaluation of the interactions among coupled services or parts of a system that cooperate to achieve a given functionality, and is also clearly a must to have. Security assessment is essential for any production service. It focuses on the runtime analysis of security-related requirements, as part of the Dynamic Application Security Testing (DAST).

Functional and Performance tests should be considered. Functional testing focuses on the evaluation of the functionality that the services exposes, leaving apart any internal design analysis or side-effects to external systems. Performance testing verifies that the service and the underlying software in execution, meets the specified performance requirements.

Operational criteria are the next part of the quality workflow for services. Obviously documentation is a must as this is an integral part of any software or service delivery. Policies are equally important. Policy documents describe what are the users expected behavior when using the service, how they can access it and what they can expect regarding privacy of their data. The documents are: Acceptable Usage Policy (AUP) and Access Policy (AP) or Terms of Use. Support is the formal way by which users and operators of the service communicate with other operators and/or developers of the service, in case of problems, and it needs to be properly documented.

Monitoring is a periodic testing of the service that needs to take place. The technology used for the monitoring is left to the developers of the underlying software to decide eventually with input from the infrastructure(s), where the service is foreseen to be integrated. In addition, quantifiable metrics should be defined to track and assess the status of services. Examples of relevant metrics are the number of users registered in the service, or using it actively. Also accounting metrics are important to track resource usage per user or group of users.

The service quality assurance is designed to test cloud services. However this does not hinder its utilization for more classical systems such as batch oriented processing in HPC systems using for instance infrastructure orchestrators [14, 24] to deploy a batch queue as a service. The process would pass by requesting the instantiation of a cluster with a pre-installation of a scheduler such as SLURM.

3. Awarding Recognition through Digital Badges

Issuing digital badges as a result of a successful quality assessment against well established criteria, is not only a visual or graphical way to show the quality of the software and services but also a way to recognize and reward compliance to best practices.

Traditional physical badges have been used as a way to prove membership or to demonstrate attaining a specific achievement. However, these types of certificate cannot satisfy the requirements for online sharing, verification, portability and the inability to be tampered. Instead, digital badges represent virtual certificates that can be easily shared, visualized and verified online. As is the case of physical badges, a digital badge still represents an achievement attained by a particular entity and demonstrates a quality seal for others to compare different entities on the basis of the quality level determined by the achieved badge.

We performed a technology scouting in order to identify the state-of-the-art responsible for issuing digital badges. The goal was to discern those that could be adopted, or even adapted, to satisfy the goal of issuing digital badges for software and for services. We focused the analysis on existing open-source tools for digital badge issuing, providing two main features: programmatic badge awarding and recipient identification aligned with the specific software/service version (e.g. release URL, commit ID, DOI).

Open Badges is the leading standard for digital credentials, originally developed by Mozilla and now managed by the IMS Global Learning Consortium[7]. In practice, an Open Badge is a PNG or SVG file that has been, by leveraging a DevOps approach, modified to adhere to the Open Badges Baking Specification which includes an iTxt section in the PNG file or some markup section inside the SVG file and can include a JSON Web Signature (JWS) for assertion.

Open Badges contain detailed metadata about achievements: who earned a badge, who issued it, and what does it mean. The data is all contained inside the badge. For each badge awarded, there is: an *Issuer Profile* describing the individual or organization awarding badges; a *BadgeClass*, the formal description of a single achievement the *Issuer* recognizes, with links to detailed criteria for how the badge may be earned; an *Assertion*, the record of an individual's achievement of the badge, with a link to evidence and expiration date.

Open Badges can be issued through a technology platform that supports the Open Badges Specification. There are several existing web-based managed platforms that have been certified by IMS Global to comply with the Open Badges specification. Among the most extended ones are *Badgr*, *Blockcerts*, *OpenBadges.me*, *OpenBadgefactory* or *MyOpenBadge*.

We selected *Badgr*[8] as the underlying platform to issue digital badges for the following reasons: originated as an open-source development, it has a fully documented API that can be used to programatically issue the Badges without any human intervention. Badgr also offers a web-based multi-tenant Graphical User Interface that can be customized and rebranded to fit the issuer needs. It is flexible in the sense that it offers different user roles (administrator vs user) to gain access to advanced functionality of the platform. Regarding authentication, Badgr offers OAuth2 Identity Provider functionality to help connected apps to securely obtain a user-specific API token to use to access badges. Finally, it also allows the definition of additional Badge designs to be tailored for several purposes. A GDPR-compliant European instance of Badgr is operated by *Instructure* as part of their Canvas Badges Europe initiative, which has been selected to issue and host the awarded badges. An example is shown in Figure 3, where the assertion is publicly available[9] and it can be independently verified via the BadgeCheck[10] platform.

To the best of our knowledge, this is the first initiative on which the Open Badges specification is being adopted not in the education field, but to recognise quality best practices for research software. Additionally the digital badges are shareable and independently verifiable by en external evaluator. For instance if multiple teams are contributing to the development of a given software product the SQAaaS architecture is such that it keeps a historic record of the series of digital badges that have been issued for that product. Also, the potential of this technology for educational purposes, for instance, as a tool to teach students best practices in programming is very big.

3.1. Mapping between quality criteria and digital badges

We defined three levels of quality that apply separately to software and services: Gold, Silver and Bronze. Table 1 shows the mapping between quality features and badge levels.

Code accessibility (QC.Acc), proper licensing (QC.Lic) and documentation (QC.Doc) are the minimum requirements to achieve the software bronze level.

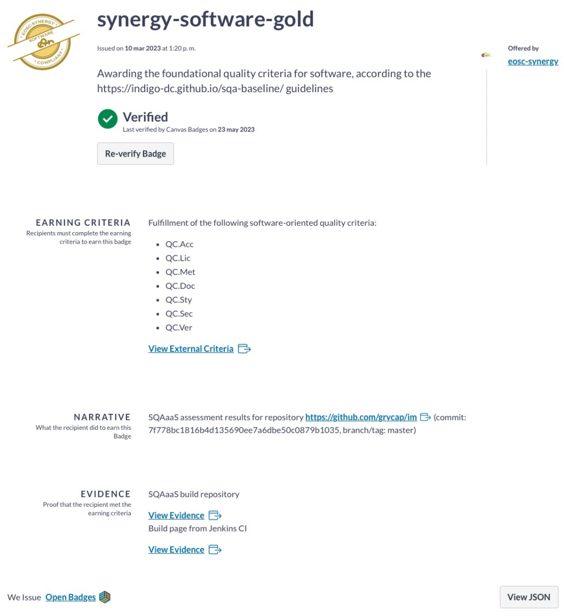
One level above, the silver level for software requires additionally a metadata file describing the software (QC.Met) and semantic versioning of the releases (QC.Ver).

[7]http://www.imsglobal.org

[8]Badgr - `https://badgr.com`

[9]Gold badge awarded for the Infrastructure Manager (IM) software: `https://eu.badgr.com/public/assertions/rkXyQH9FRj-EAMPgccz5ug`

[10]BadgeCheck - `https://badgecheck.io`



synergy-software-gold

Issued on **10 mar 2023** at 1:20 p. m.

Offered by eosc-synergy

Awarding the foundational quality criteria for software, according to the https://indigo-dc.github.io/sqa-baseline/ guidelines

Verified
Last verified by Canvas Badges on **23 may 2023**

Re-verify Badge

EARNING CRITERIA
Recipients must complete the earning criteria to earn this badge

Fulfillment of the following software-oriented quality criteria:

- QC.Acc
- QC.Lic
- QC.Met
- QC.Doc
- QC.Sty
- QC.Sec
- QC.Ver

View External Criteria

NARRATIVE
What the recipient did to earn this Badge

SQAaaS assessment results for repository https://github.com/grycap/im (commit: 7f778bc1816b4d135690ee7a6dbe50c0879b1035, branch/tag: master)

EVIDENCE
Proof that the recipient met the earning criteria

SQAaaS build repository

View Evidence

Build page from Jenkins CI

View Evidence

We Issue Open Badges

View JSON

Figure 3: An example of gold badge awarded to an open-source research software

Quality Criteria	Software Badges			Service Badges			SQAaaS
	Bronze	Silver	Gold	Bronze	Silver	Gold	Support
Accessibility (QC.Acc)	🟩	🟩	🟩				Supported
Documentation (QC.Doc)	🟩	🟩	🟩				Supported
Licensing (QC.Lic)	🟩	🟩	🟩				Supported
Code Metadata (QC.Met)		🟩	🟩				Supported
Versioning (QC.Ver)		🟩	🟩				Supported
Code Style (QC.Sty)			🟩				Supported
Security Static Analysis (QC.Sec)			🟩				Supported
Code Management (QC.Man)			🟩				
Code Workflow (QC.Wor)			🟩				
Unit Testing (QC.Uni)			🟩				Partial
Delivery (QC.Del)			🟩				
Deployment (SvcQC.Dep)				🟩	🟩	🟩	Supported
Documentation (SvcQC.Doc)				🟩	🟩	🟩	Supported
Functional Testing (SvcQC.Fun)					🟩	🟩	
Security Dynamic Analysis (SvcQC.Sec)					🟩	🟩	
API Testing (SvcQC.API)					🟩	🟩	
Integration Testing (SvcQC.Int)						🟩	Supported
Performance Testing (SvcQC.Per)						🟩	

Table 1: Summary of the Software and Services quality badges requirements per category

The gold level is achieved by software that additionally passes static security tests (QC.Sec) and source code style checks (QC.Sty). The gold level also foresees criteria that is not yet supported for automated assessment by the SQAaaS, namely code workflow (QC.Wor), unit tests (QC.Uni), and the fulfillment of both code management (QC.Man) and delivery of software artifacts (QC.Del). The unit tests criterion is supported by the SQAaaS but only for the creation of customized CI/CD pipelines as described in section 5.

Regarding services, the bronze level is achieved by those fulfilling the quality required in automated service deployment (SvcQC.Del) and documentation (SvcQC.Doc). Additionally, a positive assessment in functional testing (SvcQC.Fun), dynamic security testing (SvcQC.Sec) and API testing (SvcQC.API) grants the silver level. Successful performance (SvcQc.Per) and integration testing (SvcQc.Int) brings the gold badge. The services criteria currently supported for automated assessment by the SQAaaS includes deployment, documentation and integration testing.

4. The architecture of the Software Quality Assurance as a Service

Delivering guidelines for the accurate development of research software is not enough to engage the interest of the computational scientists. A pragmatic solution for practical quality assessment needs to be provided as well. This implies the definition of an architecture, the development of the necessary core components, high-level functionality modules, and their deployment as a production service.

The architecture of our solution, which we named *Software Quality Assurance as a Service* (SQAaaS) is depicted in Figure 4 . The ultimate goal of the SQAaaS is to cover as many criteria from the two QA baselines as possible, as long as they are suitable to be assessed in an automated fashion. The support for the criteria is planned to cover progressively the CI and CD phases of a DevOps pipeline, which match the software and service criteria, respectively. In particular this architecture enables developers to automatically check QA criteria, using CI/CD pipelines, so that each change in the source code is compliant with such practices. We expect this approach to lower the barriers of adopting such software engineering best practices.

The architecture is designed to support two main usage scenarios, or high-level functionalities. These are defined by two main modules: the Quality Assurance and Awarding (QAA) module and the Pipeline as a Service module.

The QAA module evaluates the level of compliance of a given source code repository or running service according to the previously defined SQA baselines, creates the assessment report, and awards (in case of successful qualification) quality badges [11]. The Pipelines as a Service module provides additional flexibility by enabling the composition by the end-user of custom CI/CD pipelines, based on quality criteria selected by the user.

The core software components of the SQAaaS are:

- The SQAaaS API, that exposes the features delivered by the SQAaaS, ready to be consumed by client applications.
- The SQAaaS web Portal, that facilitates the exploitation of the SQAaaS capabilities (through the SQAaaS API) by the end user.

[11] As a way to lead by example, all the SQAaaS core components have been validated by the QAA module

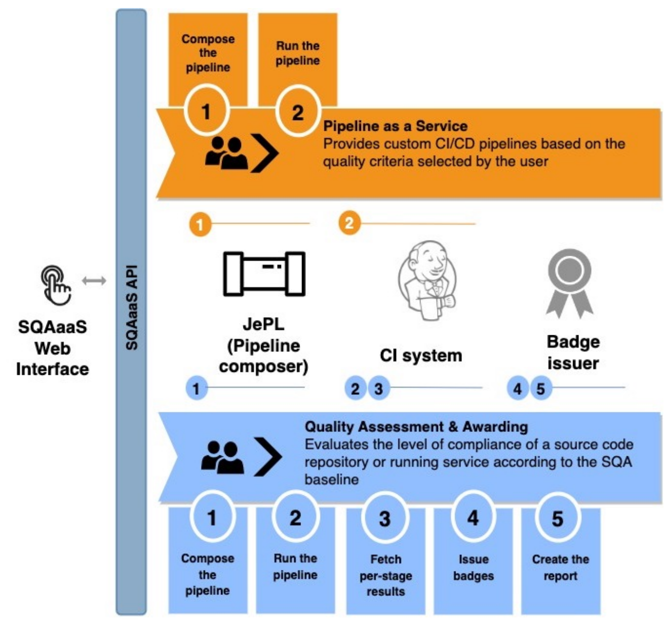

Compose the pipeline
Run the pipeline
Pipeline as a Service
Provides custom CI/CD pipelines based on the quality criteria selected by the user
SQAaaS Web Interface
SQAaaS API
JePL (Pipeline composer)
CI system
Badge issuer
Quality Assessment & Awarding
Evaluates the level of compliance of a source code repository or running service according to the SQA baseline
Compose the pipeline
Run the pipeline
Fetch per-stage results
Issue badges
Create the report

Figure 4: High Level architecture of the Software Quality Assurance as a Service

- The CI system, that provides to the platform automatic provision capabilities of the computing resources, and execution of the CI/CD pipelines. Jenkins CI is the natural choice since the Jenkins Pipeline as Code technology fully addresses the SQAaaS requirements with regards to the programmatic composition of the CI/CD pipelines with code. Also, the Jenkins API exposes the functionality needed for the pipeline building management, such as triggering builds or returning the status of the execution.
- The Jenkins Pipeline Library (JePL), that enables easier composition of the CI/CD pipelines according to the quality baselines. These pipelines are then executed by the Jenkins CI.
- The Badge issuer, implemented by Bagdr, that based on the QAA module results issues digital badges highlighting the quality achievements of the assessed software or service.

The following sections describe in greater detail the high-level capabilities of the platform and the development of the core software components.

5. High Level functionalities of the SQAaaS

The SQAaaS platform provides the tools that facilitate the adoption of good software development practices. Hence, quality is considered starting at the code level, and validated whenever any change is done through the execution of workflows that encompass a series of checks, which are commonly known as CI/CD pipelines. There are two main high level capabilities arising from the architecture: *Quality Assessment and Awarding* (QAA), and *Pipeline as a Service*.

5.1. Quality Assessment and Awarding module

The SQAaaS platform analyzes the level of compliance of a given code repository with the set of quality baselines as described in section 2. Following this assessment, it also certifies the results obtained by issuing digital badges when a minimum set of quality achievements is reached. With this information the QAA module produces two main outcomes:

- A quality report where the results of the assessment are described for all the supported quality criteria. The validity of each quality criterion is computed according to the outputs provided by the tools employed to do the assessment of each criteria.
- A digital badge highlighting the quality achievements of the software or service according to the mapping shown in Table 1.

The QAA module operation leverages the pipelines created by the JePL library to conduct the assessment. The JePL library is one of the core developments in this work (see section 6 for a detailed description).

5.1.1. The quality assessment process

For any given source code repository the QAA module performs the quality assessment in terms of the criteria defined in the quality baselines. In practice, for each quality criterion the QAA module walks through all its subcriteria and identifies the right open source software tool to perform the corresponding quality assessment.

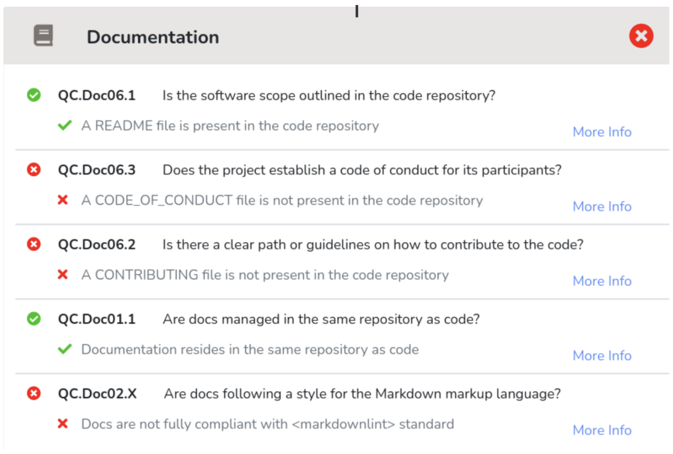
This selection is done by the QAA through the *tooling metadata* component, one of the core developments in this work (see section 6). In general, multiple tools might be needed for an individual criterion, given that each one contains additional subcriteria that cannot be covered with a single tool. For instance, to check the code style (QC.sty) in the case of python codes we use *tox*, *flake8* and *pycodestyle*.

5.1.2. The quality assessment report and badge issuing

The QAA processing consists in iterating over the set of stages of the quality assessment pipeline, selecting and running the plugins for the output validation, and finally collecting all the returned data. This data is formatted according to the SQAaaS API specification so that the API clients, such as the SQAaaS web portal, can render and present it as the final outcome.

Figure 5 shows an excerpt from a QAA report as rendered by the SQAaaS portal. Both the SQAaaS API and the SQAaaS portal are core developments of this work (see sections 6 and 7 respectively).

The final report presents a validation per criterion based on the subcriteria validation. The validity of each criterion is estimated from the individual results of the subcriteria, which also takes into account their level of criticality. As an example, being QC.Sty01 a mandatory subcriterion (MUST level) and QC.Sty02.2 an optional one (SHOULD level), if QC.Sty02.2



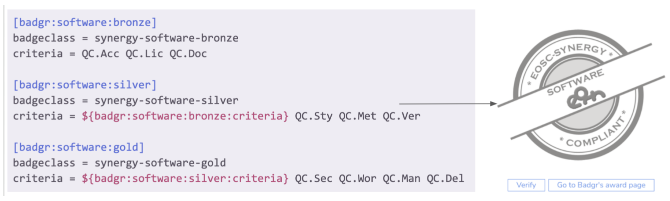
Documentation

✓ QC.Doc06.1	Is the software scope outlined in the code repository?	
	✓ A README file is present in the code repository	More Info
✗ QC.Doc06.3	Does the project establish a code of conduct for its participants?	
	✗ A CODE_OF_CONDUCT file is not present in the code repository	More Info
✗ QC.Doc06.2	Is there a clear path or guidelines on how to contribute to the code?	
	✗ A CONTRIBUTING file is not present in the code repository	More Info
✓ QC.Doc01.1	Are docs managed in the same repository as code?	
	✓ Documentation resides in the same repository as code	More Info
✗ QC.Doc02.X	Are docs following a style for the Markdown markup language?	
	✗ Docs are not fully compliant with <markdownlint> standard	More Info

Figure 5: An excerpt of the final report provided by the QAA module. This image shows only the results from the Documentation (QC.Doc)-related sub-criteria.

succeeds but QC.Sty01 fails to validate, then the QAA will report the parent criterion QC.Sty as failed.

Building on the gathering of the reporting data the QAA module is also able to issue digital badges. The SQAaaS API defines a configurable mapping between criteria and associated badges as defined in 1. The SQAaaS API iterates over the defined classes (see example in Figure 6) to obtain the appropriate class. Whenever the fulfilled criteria matches a given badge class, the corresponding digital badge is issued and included as part of the QAA report, as shown in Figure 6.



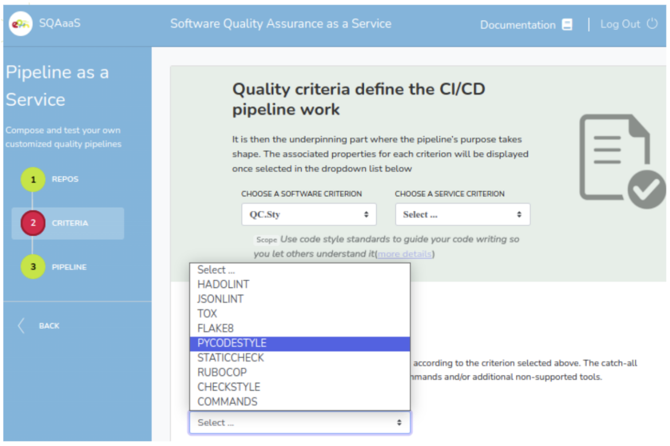
```
[badgr:software:bronze]
badgeclass = synergy-software-bronze
criteria = QC.Acc QC.Lic QC.Doc

[badgr:software:silver]
badgeclass = synergy-software-silver
criteria = ${badgr:software:bronze:criteria} QC.Sty QC.Met QC.Ver

[badgr:software:gold]
badgeclass = synergy-software-gold
criteria = ${badgr:software:silver:criteria} QC.Sec QC.Wor QC.Man QC.Del
```

Figure 6: Mapping between criteria and corresponding badges as it is defined for the SQAaaS API. The right side shows a silver badge issued for a software product, as shown in the SQAaaS portal

5.2. *Pipeline as a Service module*

The most high-level interface, the SQAaaS web portal, offers end-users the possibility to create graphically customized pipelines. This capability is achieved through the Pipeline as a Service module.

Through this module, the user selects the set of quality criteria and corresponding individual (open source) tools that will be executed to verify and validate the software (see Figure 7). Each tool allows the fulfilment of one or several of the good practices, and might require additional work, such as for example writing unit tests. In contrast, other tools just parse the code or run the software in order to find commonly known bad practices (e.g. related to style or security).



SQAaaS
Software Quality Assurance as a Service
Documentation
Log Out
Pipeline as a Service
Compose and test your own customized quality pipelines
1 REPOS
2 CRITERIA
3 PIPELINE
BACK

Quality criteria define the CI/CD pipeline work

It is then the underpinning part where the pipeline's purpose takes shape. The associated properties for each criterion will be displayed once selected in the dropdown list below

CHOOSE A SOFTWARE CRITERION: QC.Sty

CHOOSE A SERVICE CRITERION: Select ...

Scope *Use code style standards to guide your code writing so you let others understand it*(more details)

Select ...
HADOLINT
JSONLINT
TOX
FLAKE8
PYCODESTYLE
STATICCHECK
RUBOCOP
CHECKSTYLE
COMMANDS

according to the criterion selected above. The catch-all
mands and/or additional non-supported tools.

Select ...

Figure 7: Capture of the Pipeline as-a-Service graphical interface with the end user

Using the Pipeline as a Service the developer (end-user) can create pipelines based on the available quality criteria, and obtain the results from the pipeline execution. This module also allows the end-user to download the composed pipeline. Automation enabling a pull request to automatically add the resulting pipeline to a source code repository is also provided.

The realization of the Pipeline as a Service module required the development of a pipeline composer, which implements the support for the criteria established in the quality baselines through the composition of CI/CD pipelines. The module also makes use of the CI system, SQAaaS API and SQAaaS web Portal.

6. SQAaaS core components development

This section describes the core software developments to implement the architecture of the SQAaaS. Figure 4 identifies the set of components that take part in each high-level capability described in the previous section. We identify two main sets of components, namely the existing services integrated and leveraged by the platform, and the base components developed from scratch to enable the core capabilities of the platform.

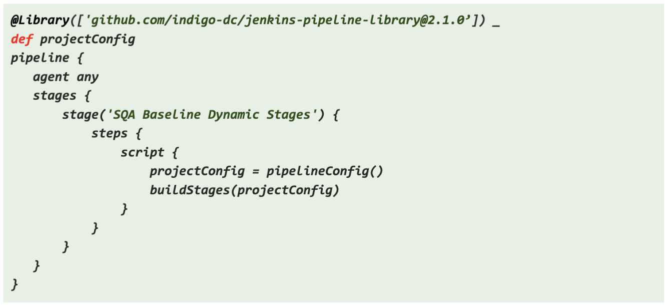
The first set is comprised by open source technologies adopted to fulfill a specific requirement. The SQAaaS platform relies on the Jenkins CI system[12] to manage the programmatic execution of CI/CD pipelines in a scalable manner. On the other hand, as part of the awarding feature, the Open Badges-compliant service Badgr[13] is used to issue digital badges.

Throughout this section we will focus on the core components grouped under the second set. Let us first describe the three main design choices made: YAML-based CI/CD pipelines, API-first (contract-based API), and Static website.

1. YAML provides a human-readable language representation for data structures which is often used to write configuration files. Based on its readability, YAML is commonly a better choice than alternative options like JSON,

[12]https://jenkins.io
[13]https://eu.badgr.com/

and at the same time it can benefit from the consistency of the latter, leveraging the JSON schema validation. The JePL library only requires dealing with YAML files. Therefore, JePL is also perfectly suitable to be used directly by the end user without the need of relying on the SQAaaS platform.
The only required file that is not written in YAML is the Jenkinsfile. An example of the typical content is shown in Figure 8.



```
@Library(['github.com/indigo-dc/jenkins-pipeline-library@2.1.0’]) _
def projectConfig
pipeline {
   agent any
   stages {
       stage('SQA Baseline Dynamic Stages') {
           steps {
               script {
                   projectConfig = pipelineConfig()
                   buildStages(projectConfig)
               }
           }
       }
   }
}
```

Figure 8: Content of the Jenkinsfile as required by JePL library

2. The SQAaaS API has been designed using the OpenAPI v3.0.3 specification [14]. The OpenAPI specification uses a human-consumable language that establishes a contract to stipulate its behaviour. Accordingly, the design of the API is more accurate as it implies a collaborative effort (from both technical and non-technical parties) in order to define what will be subsequently implemented with code. This approach, known as API-first or contract-based API, was adopted for the SQAaaS API. The API documentation is the fundamental tool and vehicle for consumer applications. This is the case for example of the SQAaaS Web. Accurate API documentation is fundamental for a smooth integration once the real API is available.
3. The SQAaaS web portal has been instantiated as a static website employing a JAMStack[15] approach, exclusively incorporating HTML, CSS, and JavaScript. The choice of the JAMStack framework stems from its features in terms of security, scalability, performance, and portability, rendering it particularly apt for our objectives. In the construction of the front-end, Vue.js was employed due to its intuitive support for user interface development and the vibrancy of its community. Furthermore, Vue.js demonstrated facile integration capabilities with pre-existing applications.

6.1. JePL Library

The JePL library[16] is a CI/CD pipeline composer to easily configure the SQAaaS pipelines. The library can be used without knowing the Jenkins Pipeline as Code syntax. In this sense, a JePL pipeline defines the validation workflow following the quality criteria defined in the quality baselines in section 2.

This library leverages the YAML language to describe the criteria of the QA baselines to be assessed. In practice, a single configuration file, the `config.yml` file, describes the required configurations and stages for the criteria evaluation. The additional configuration files are required to manage the services needed to execute the CI checks. These include a minimal Jenkins PaC definition or Jenkinsfile and `docker-compose.yml`. The Jenkinsfile has the common base code using the Groovy language to load the JePL Jenkins shared library. The project-related information is in `config.yml` and `docker-compose.yml`. Thus, software, services and tests are represented in these files.

The core value of the JePL is that dynamically generates the set of Jenkins pipeline stages that will validate the QA criteria defined in the `config.yml` file. The UML diagram from Figure 9 summarizes the JePL implementation.

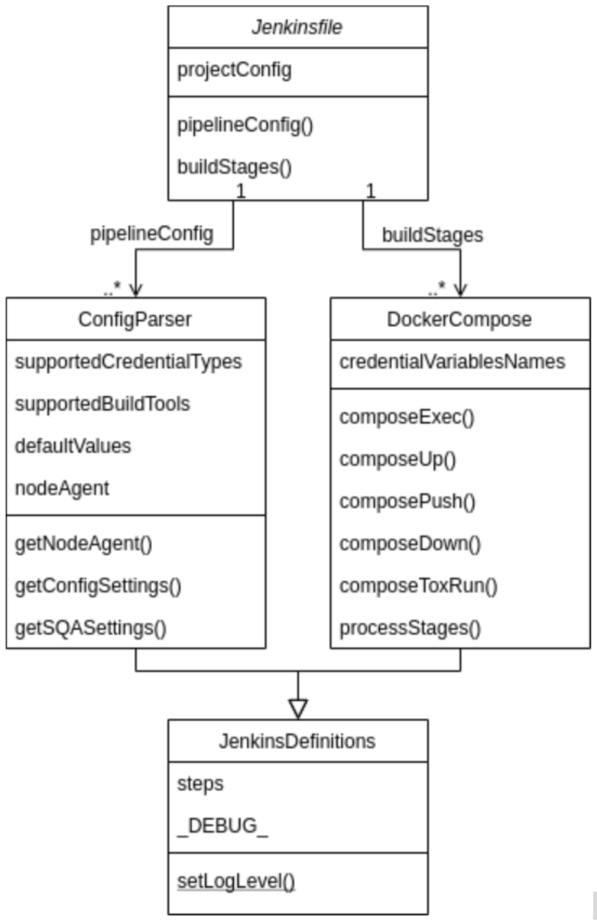


Figure 9: UML diagram with a short representation of the JePL main classes

The `Jenkinsfile` loads and initiates JePL by calling the Groovy closures `pipelineConfig()` and `buildStages()`. The SQA criteria definitions placed in the `config.yml` file are loaded and validated with the call to `pipelineConfig()`. Similarly, the `docker-compose.yml` file configurations are loaded with the call to `buildStages()`. This closure also receives the

[14] `https://spec.openapis.org/oas/v3.0.3`
[15] `https://jamstack.org/`
[16] Code available at https://github.com/indigo-dc/jenkins-pipeline-library/

pipeline definitions returned from `pipelineConfig()` and generates the corresponding stages.

The user must specify the parameters for each criterion, such as the source code repositories and the tools to be used. Every pipeline execution will perform a new deployment of the required software environment using Docker Compose. When the job execution finishes (either succeeding or failing) the allocated resources are released.

The Jenkins Controller dispatches each pipeline job to a Jenkins Agent and maps the environment requirements for Docker Compose orchestration. The Jenkins Agents are responsible for creating the required environments over any given cloud computing platform, either on-premises (e.g. OpenStack) or public. Figure 10 shows the Jenkins geo-distributed architecture capability, where multiple agents can be located on different clouds.

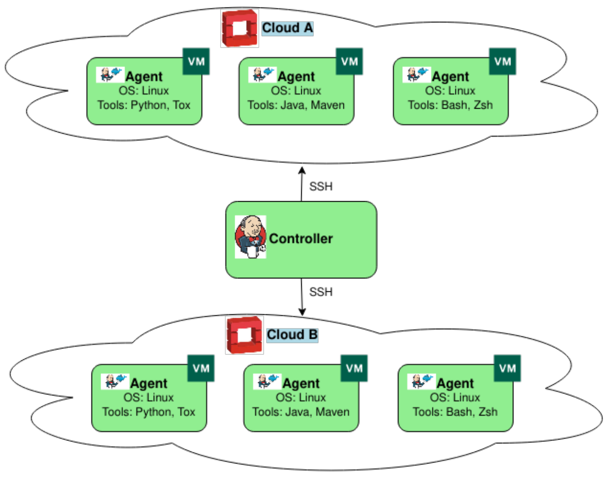


Figure 10: Jenkins Controller and Agent interaction between multiple Cloud sites

6.2. *Tooling metadata component*

Each quality criterion establishes the requirements and good practices for a specific quality characteristic. The SQAaaS relies both on internal checks and open-source tools to cover all the requirements from the defined quality criteria. The tooling metadata component maintains the full characterisation of each tool and includes the contextual and operational metadata required to define and execute the JePL pipelines.

The tooling metadata is a JSON-formatted document containing the mapping of each criterion - subcriterion - tool, so that the QAA can compose a fully-fledged JePL pipeline that will carry out the quality assessment process. This JSON document is stored and maintained through a public code repository so it can be fetched programmatically (and remotely) by the SQAaaS API. Therefore, the set of supported tools by the SQAaaS platform are fully customizable, and any new tool or change in the existing definition will be dynamically loaded and readily available in the SQAaaS portal.

Each tool requires an appropriate environment for execution in the CI system, which includes the tool itself with the related dependencies. Environment information for each tool is available in the tooling metadata, through the docker property. This information includes primarily the location of the remote image from a Docker registry, or the Dockerfile, which must be available locally in the same repository as the tooling metadata. The docker property is provided as input to the QAA module to create the Docker Compose file used by JePL.

An example of tooling metadata is shown in Figure 11. The args property in the tools' definition provides the arguments that will be used to compose the commands to be executed in the pipelines. The type of argument can fall into the three categories set out below:

- *subcommand*: many tools break up their functionality into subcommands. One popular example is the git tool that provides multiple subcommands (e.g. git add, git commit, ..).
- *positional*: those arguments that are required and that are defined only by their value. They can be used both with a command or a subcommand. Continuing with the example above, the git add subcommand always requires a positional argument (e.g. git add file1).
- *optional*: those arguments that might be provided, but they are not required. The option name, which contains a single dash for the short version and two dashes for the long version, can be used both in conjunction with a value or, otherwise, by itself. An example is git add –verbose file1.

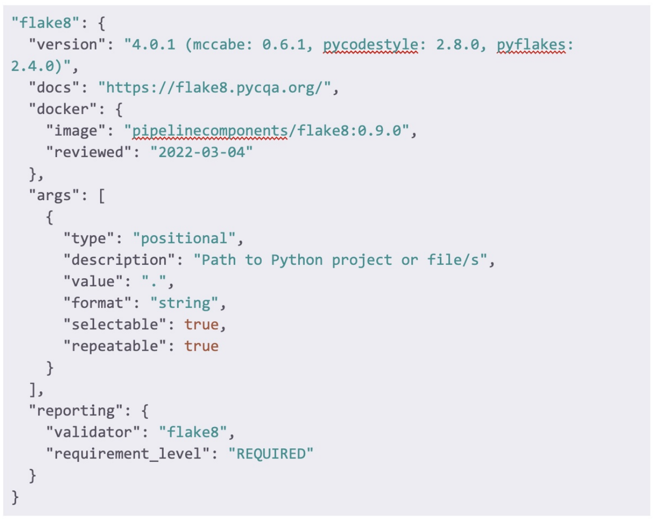
Additional properties describe what the argument does (description), provide a default value (value) or, in the case of the optional arguments, define the option name (option). Furthermore, there are properties specific for SQAaaS API clients, such as the SQAaaS web:

- format, which identifies the type of data (e.g. string or array)
- selectable, whether the property shall be showed (true) or hidden (false)
- repeatable, whether the property can be provided multiple times

6.2.1. *The reporting property*

Once the QAA pipeline has been executed and the logs are available for each tool, the SQAaaS API relies again on the tooling metadata to get the appropriate plugin for tackling the output of each tool. The reporting property defines the mapping between the tool and the plugin through the *validator key*. Furthermore, this property contains additional information that might be passed to the plugin at runtime. Figure 11 provides an example of how the reporting property is defined.

More technical details about the tooling metadata, together with a list of the open-source tools used in the SQAaaS platform, are available in the online documentation [25].



```
"flake8": {
  "version": "4.0.1 (mccabe: 0.6.1, pycodestyle: 2.8.0, pyflakes:
2.4.0)",
  "docs": "https://flake8.pycqa.org/",
  "docker": {
    "image": "pipelinecomponents/flake8:0.9.0",
    "reviewed": "2022-03-04"
  },
  "args": [
    {
      "type": "positional",
      "description": "Path to Python project or file/s",
      "value": ".",
      "format": "string",
      "selectable": true,
      "repeatable": true
    }
  ],
  "reporting": {
    "validator": "flake8",
    "requirement_level": "REQUIRED"
  }
}
```

Figure 11: Tool (flake8) excerpt from the tooling metadata. The metadata includes several chunks of information, such as the tool arguments (args), runtime information (docker) and the output validation-related part (reporting)

6.3. *SQAaaS API*

The API server or SQAaaS API constitutes the cornerstone of the SQAaaS platform as it glues together all the different components in the architecture. This includes leveraging the features of the JePL library to create CI/CD pipelines, interfacing with the APIs provided by the upstream services, such as the CI system and the badge issuer, and conducting the communication with the API clients, in particular the SQAaaS web interface.

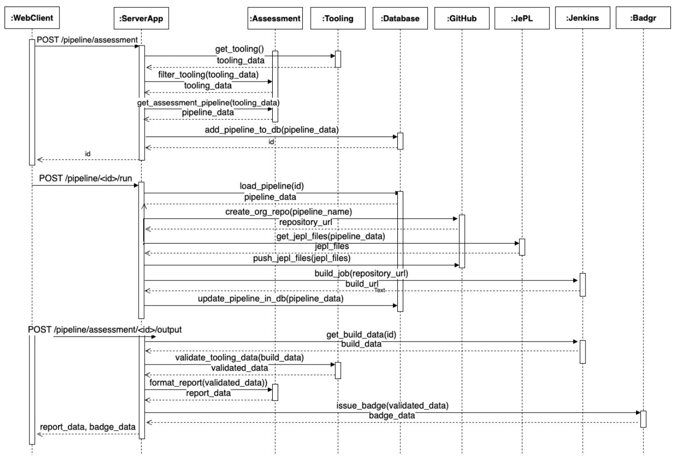

:WebClient
:ServerApp
:Assessment
:Tooling
:Database
:GitHub
:JePL
:Jenkins
:Badgr
POST /pipeline/assessment
get_tooling()
tooling_data
filter_tooling(tooling_data)
tooling_data
get_assessment_pipeline(tooling_data)
pipeline_data
add_pipeline_to_db(pipeline_data)
id
id
POST /pipeline/<id>/run
load_pipeline(id)
pipeline_data
create_org_repo(pipeline_name)
repository_url
get_jepl_files(pipeline_data)
jepl_files
push_jepl_files(jepl_files)
build_job(repository_url)
build_url
Text
update_pipeline_in_db(pipeline_data)
POST /pipeline/assessment/<id>/output
get_build_data(id)
build_data
validate_tooling_data(build_data)
validated_data
format_report(validated_data))
report_data
issue_badge(validated_data)
badge_data
report_data, badge_data

Figure 12: Sequence diagram with the full execution workflow of the SQAaaS API for quality assessments

The SQAaaS API implements and exposes to the outside world the two high-level capabilities described in the previous section. All the features implemented are described through a contract-based approach using the OpenAPI specification[17].

The main operation path enables the realisation of the standard operations that can be performed with a CI/CD pipeline, such as creating and cancelling, as well as running pipelines or checking the status.

Figure 12 depicts the sequence of interactions during the operation of a quality assessment. This operation combines the use of the aforementioned API paths, which are triggered by the web client as HTTP requests. Internally, as it can be seen in the figure, these requests lead to a series of actions that imply interfacing with the core components of the SQAaaS platform and upstream services such as the CI server, the repository platform (GitHub) and the digital badge issuer (Badgr).

The API server is the conductor of the SQAaaS platform. It is responsible for interacting with the previously described tooling component in order to get the appropriate set of tools that take part in each pipeline. The tool selection process is derived from the inspection of the file extensions that exist in the code repository, so that only the tools that are capable of dealing with the existing file formats are considered. The set of filtered tools are then grouped by stages, where each represent a different quality characteristic. For defining the CI/CD pipeline, the SQAaaS API leverages the JePL library, which results in the composition of the file structure seen in section 6.

In order to perform a quality assessment, the SQAaaS API deals with the additional upstream services, through their corresponding APIs. The CI/CD pipeline, in JePL format, produced in the previous step is then added to a code repository in the GitHub platform. This operation will trigger a (push) event so that the Jenkins CI service gets notified and starts the execution of the CI/CD pipeline. Through the Jenkins API, the SQAaaS API detects when the pipeline finished executing and, subsequently, extracts and validates the assessment data. This is a complex process since it implies parsing all the outputs from the set of quality assessment tools in the pipeline, and assert their validity. With this information, the SQAaaS API is able to elaborate a quality report describing the quality criteria being fulfilled. This sets the ground for the digital badge issuing process, where the SQAaaS API relies on the mapping between quality criteria and badges Table 1 to issue a matching badge using the Badgr API.

7. User Interface to quality assessment

The SQAaaS back-end components are exposed to the end user via the SQAaaS Portal, a web-based graphical user interface that delivers the functionality of both the quality assessment and awarding, and the pipeline as a service modules.

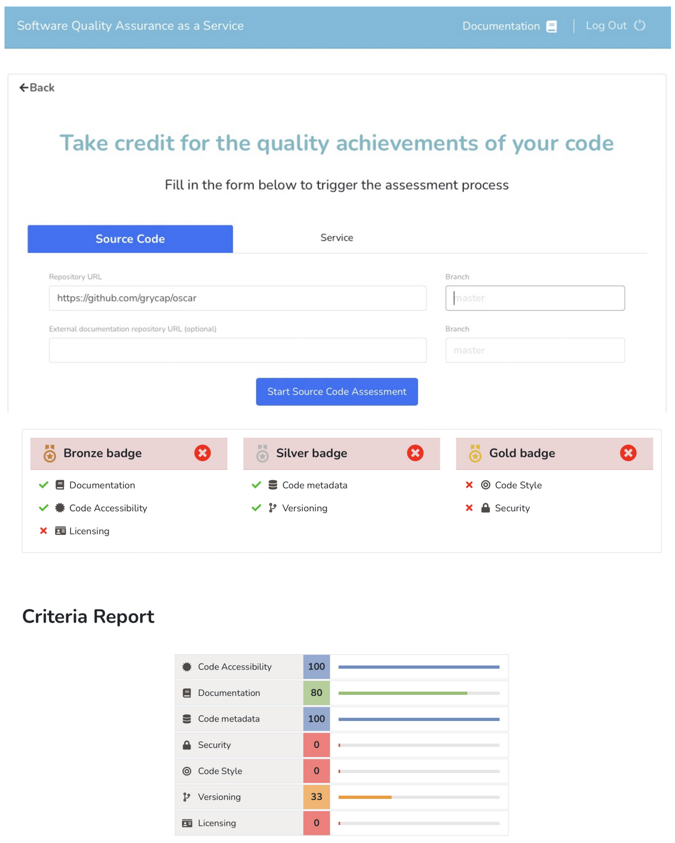
This is an open-source development[18] that is publicly offered as a managed service[19] so that end users can assess the quality of their research software products and, if required, build their own customized pipelines for software testing.

[17]SQAaaS API specification: `https://github.com/eosc-synergy/sqaaas-api-spec`

[18]SQAaaS-Web GitHub repository: `https://github.com/eosc-synergy/sqaaas-web`

[19]SQAaaS-Web: `https://sqaaas.eosc-synergy.eu`

The SQAaaS Portal is offered in a serverless manner via GitHub pages to reduce maintenance and enhance scalability of the front-end, which runs entirely in the user's web browser via a JAMStack development approach. It is integrated with EGI Check-In[20], a proxy service that acts as a central hub to connect federated identity providers (IdPs). Therefore, users can login automatically into the service not only via social network identities, but also via institutional accounts where applicable (universities, research centers, etc.)


Software Quality Assurance as a Service
Documentation
Log Out
Back
Take credit for the quality achievements of your code
Fill in the form below to trigger the assessment process
Source Code
Service
Start Source Code Assessment
Criteria Report

Repository URL	https://github.com/grycap/oscar
Branch	master
External documentation repository URL (optional)	
Branch	master

Bronze badge ✗
- ✓ Documentation
- ✓ Code Accessibility
- ✗ Licensing

Silver badge ✗
- ✓ Code metadata
- ✓ Versioning

Gold badge ✗
- ✗ Code Style
- ✗ Security

Code Accessibility	100
Documentation	80
Code metadata	100
Security	0
Code Style	0
Versioning	33
Licensing	0

Figure 13: Screenshots of the web-based SQAaaS platform

The SQAaaS Portal can render dynamically the tools defined in the tooling metadata. This data is received from the API and renders the tools according to the quality criterion they fall into. The tool's arguments are also displayed so that the user can set custom values for them. The type of web component that is required for each argument is also defined in the metadata, and thus, the SQAaaS Portal knows how to render them. This means that any change in the tooling JSON's metadata will be automatically reflected on the web without changes to the source code.

Figure 13 shows the aspect of the wizard-like interface that guides the end users to perform the automated assessments. The colour-code scheme provides timely visual feedback concerning compliance with the criteria of the different levels of badges. This is supplemented with a detailed explanation for each criterion so that the product owner can plan a path for compliance to improve the product quality, meet the criteria, and obtain the corresponding badge.

8. SQAaaS business logic

Regarding the expected functionality of the SQAaaS API (Quality Assessment and Pipeline as a Service), the following paths have been implemented:

- `/pipeline`
  - `POST`: creates a new pipeline definition in the database
  - `GET`: returns the available pipelines from the database
- `/pipeline/<id>` `GET`: returns the definition for the given pipeline
- `/pipeline/<id>/config` `GET`: returns the JePL main config definition
- `/pipeline/<id>/composer` `GET`: returns the JePL composer definition
- `/pipeline/<id>/jenkinsfile` `GET`: returns the JePL *Jenkinsfile* definition
- `/pipeline/<id>/run` `POST`: runs the pipeline in the Jenkins endpoint
- `/pipeline/<id>/status` `GET`: obtains the execution status of the pipeline in Jenkins
- `/pipeline/<id>/compressed_files` `GET`: returns a compressed file format with the JePL files structure
- `/pipeline/<id>/pull_request` `POST`: create a pull request in an upstream repo (provided by the user) in order to add the generated JePL files.

Figures 14 and 15 summarize the business logic for the most relevant API paths. It is important to note the **use of the GitHub API in order to trigger the builds in the Jenkins CI system**. In particular, the new pipelines are run by Jenkins as a result of their presence in GitHub repository organization, and thus, they are not being directly composed leveraging Jenkins API. The main rationale behind this approach is to benefit from a Version Control System (VCS) approach in the maintenance of the pipelines. Thus, each pipeline modification is versioned, and subsequently executed in Jenkins, granted by a previous integration step among the two systems.

Figure 14 showcases this approach, where the execution of a pipeline involves the creation of the repository (in the relevant GitHub repository) with the JePL structure [`create_org_repo()`, `push_jepl_files()`] and, afterwards, it triggers the organization scan [`scan_organization()`] so that the new repository (or changes thereof) are detected by Jenkins and the pipeline can be ran. If the pipeline has already run and the repository

[20]EGI Check-in: `https://aai.egi.eu/registry/`

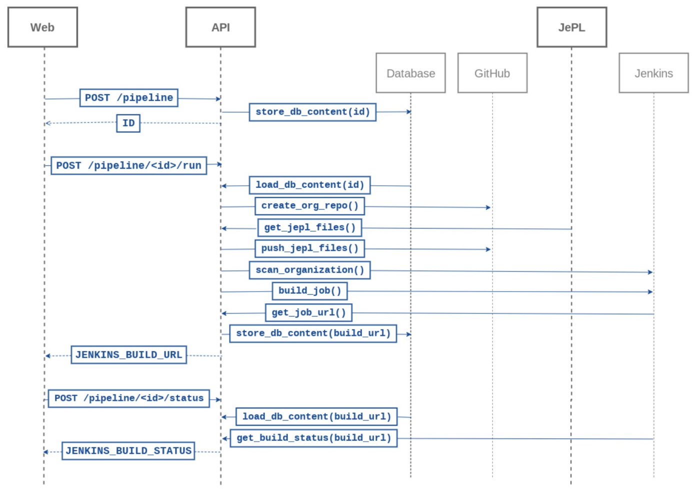


Figure 14: Sequence diagram for the API paths that i) create, ii) run, and iii) get the status (in Jenkins) of a pipeline created with the JePL component.

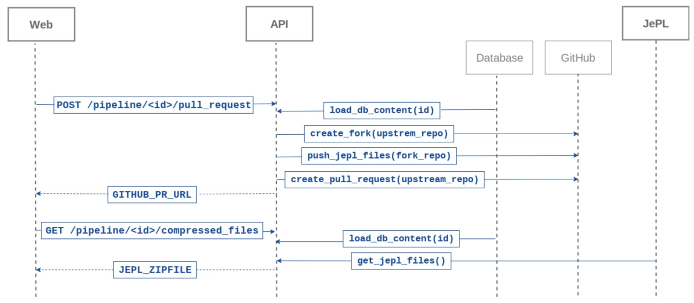

Web
API
JePL
Database
GitHub
POST /pipeline/<id>/pull_request
load_db_content(id)
create_fork(upstrem_repo)
push_jepl_files(fork_repo)
create_pull_request(upstream_repo)
GITHUB_PR_URL
GET /pipeline/<id>/compressed_files
load_db_content(id)
get_jepl_files()
JEPL_ZIPFILE

Figure 15: Sequence diagram for the API paths that i) create a PR in GitHub, and ii) get the JePL file structure as a compressed (ZIP) file.

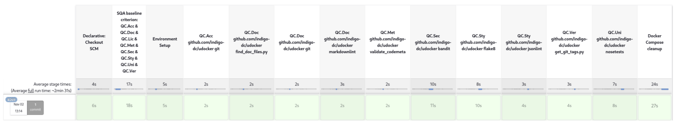
exists, the pipeline is triggered directly through the Jenkins API [`build_job()`].

Figure 15 showcases the workflows associated with two important features. Once the pipeline is composed, stored and executed, these two API paths provide the means to make the pipeline available in the user own code repository, either by creating a PR to such a repository or by downloading the (JePL) pipeline files in a compressed format. In the former case, the SQAaaS API uses again the relevant GitHub organisation to create a fork of the target repository in order to create a PR without the need of handling user credentials.

9. Validation of the SQAaaS platform

There are several usage patterns of the SQAaaS platform, ranging from educational purposes, to support development processes or the independent testing of best practices in a given repository for external evaluation. This section describes the usage of the platform to support testing and code development.

As an example, we use the evolution of the quality assess-



	Declarative: Checkout SCM	SQA baseline criterion: QC.Acc & QC.Doc & QC.Lic & QC.Met & QC.Sec & QC.Sty & QC.Uni & QC.Ver	Environment Setup	QC.Acc github.com/indigo-dc/udocker git	QC.Doc github.com/indigo-dc/udocker find_doc_files.py	QC.Doc github.com/indigo-dc/udocker git	QC.Doc github.com/indigo-dc/udocker markdownlint	QC.Met github.com/indigo-dc/udocker validate_codemeta	QC.Sec github.com/indigo-dc/udocker bandit	QC.Sty github.com/indigo-dc/udocker flake8	QC.Sty github.com/indigo-dc/udocker jsonlint	QC.Ver github.com/indigo-dc/udocker get_git_tags.py	QC.Uni github.com/indigo-dc/udocker nosetests	Docker Compose cleanup
Average stage times: (Average full run time: ~2min 31s)	4s	17s	5s	2s	2s	2s	3s	2s	10s	8s	3s	3s	7s	24s
#269 Nov 02 13:14 1 commit	6s	18s	5s	2s	2s	2s	2s	2s	11s	10s	4s	4s	8s	27s

Figure 16: Jenkins pipeline created with the SQAaaS Pipeline as a Service for *udocker*

ment for *udocker*[21], a user-level tool to execute Docker containers in user space without requiring root privileges [26]. The first production version of *udocker* dates back to 2018, and the most recent release is from July 2023.

Table 2 shows the evolution of the quality metrics with the release tags. The quality assessment process proves to be very effective as a support tool for the evolution of the software stack. The case of *udocker* is representative of state-of-the-art software from the point of view of software engineering. In particular, the most basic features such as licensing, basic versioning or basic documentation were already fulfilled in the first version of *udocker*. The basic documentation criterion was also fulfilled from the very beginning. However, being fully compliant with a markdown standard has taken more effort in the development process and has been achieved only from release candidate $v1.3.5 - rc.4$ on.

The inclusion of metadata in the code repository is a recent evolution of software engineering standards and, therefore, is not present in many software repositories. In the case of *udocker* this feature was included from release 1.3.4. The SQAaaS tool has proven to be very effective in detecting possible static security issues, that were fixed as they were detected from version 1.1.8 on. Code style, and the evolution of *udocker* to Python 3 (starting from release 1.3.0) has generated quite an amount of work due to the changes from Python 2. The automatic testing of the new code style requirements has been a very useful tool to support the developers' work towards a coherent migration between the two versions of Python.

Besides enabling the developers to improve the quality of the software, the SQAaaS tool has also been essential to keep the level of quality for the releases. The gold badge for software was obtained for release 1.3.5 and was held for subsequent releases. To this end, the Pipeline as a Service module was also adopted by the developers that created a complete pipeline including the unit tests criterion. Figure 16 shows the outcome of the pipeline that is automatically triggered for each pull to the master branch.

The SQAaaS platform has been deployed in production as a service, freely available online, which is being used regularly by hundreds of users around the world[22]. The statistics collected using Google Analytics indicate the potential impact of this development. The SQAaaS platform has performed over 2800 software assessments, more than 300 unique repositories, and has awarded >125 Digital Badges.

[21] `https://github.com/indigo-dc/udocker`

[22] See `https://www.eosc-synergy.eu/live-statistics-sqaaas/` for live statistics

The software stacks that have been assessed range from system software middleware tools, infrastructure managers, workflow managers, to heavily used scientific applications in the areas of Earth Sciences (eg. Fall3D), Bio-Informatics (eg. haddock prodigy) or High Energy Physics (eg. openQCD) [23]. Notice that the components themselves of the SQAaaS service have also been assessed for the sake of consistency.

10. Conclusions and future work

The purpose of the SQAaaS has been providing easy to use tools for software developers to evaluate their work ensuring software quality by following the DevOps culture, good practices and enabling the adoption of Continuous Integration and Continuous Delivery. In this sense the two major outcomes can be summarized as: i) ease the adoption of CI/CD workflows or pipelines in research software environments through the implementation of different interfaces (graphically through the portal, programmatically through the API and via YAML documents using the JePL library), and ii) add transparency to the quality-related process used during the software development life cycle through the usage of a quality assessment module that based on the quality baselines has the extended capability of issuing digital badges according to well-defined quality achievements.

All platform components are open source, free to deploy anywhere. The solution is highly modular and allows to use any tool or platform, not only those we are supporting in the release that makes the basis of this publication. It is easy to use through a web interface, which decreases the learning curve of potential customers. The API is open source, well documented and easy to extend for new tools or platforms. It provides recognition and quality stamping, thanks to the digital badges issuance, together with a detailed quality report.

While performing these developments multiple promising paths for improvement have been open, which constitute the basis for evolution and future work. Among them increasing the coverage of programming languages, including others that are becoming popular in research such as Julia, increase coverage of git-based social coding platforms (currently it supports GitHub and GitLab). The inclusion of additional tools suitable for automated static and dynamic security analysis, including the implementation of Artificial Intelligence tools, are also a promising evolution to be included in a future release of the SQAaaS platform.

Acknowledgements

This work has taken place in the framework of the project EOSC-Synergy funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 857647.
The work also benefited from the support of INCD funded by FCT and FEDER under the project 01/SAICT/2016 nº 022153.

Version	**1.0.0**	**1.1.1**	**1.1.8**	**1.3.1**	**1.3.4**	**1.3.5-rc.2**	**1.3.5-rc.3**	**1.3.5-rc.4**	**1.3.5-rc.5**	**1.3.5**
Badge / QC	None	None	None	None	None	None	SILVER	SILVER	GOLD	GOLD
QC.Acc	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Acc01 MUST	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Doc							Ok	Ok	Ok	Ok
QC.Doc01.1 MAY	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Doc02.X MAY	(3)	(3)	(3)	(3)	(3)	(3)	(3)	Ok	Ok	Ok
QC.Doc06.1 MUST	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Doc06.2 MUST	(5)	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Doc06.3 MUST	(4)	(4)	(4)	(4)	(4)	(4)	Ok	Ok	Ok	Ok
QC.Lic	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Lic01 MUST	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Lic01.1 MUST	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Lic02 MUST	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Lic02.1 MAY	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Met					Ok	Ok	Ok	Ok	Ok	Ok
QC.Met01 SHOULD	(7)	(7)	(7)	(8)	Ok	Ok	Ok	Ok	Ok	Ok
QC.Sec			Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Sec02 MUST	(6)	(6)	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Sty									Ok	Ok
QC.Sty01 MUST	(1)	(1)	(1)	(1) (9)	(1)	(1)	(1)	(1)	Ok	Ok
QC.Ver	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Ver01 SHOULD	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Ver01.0 MUST	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Ok
QC.Ver02 SHOULD	(2)	(2)	(2)	Ok	Ok	(2)	(2)	(2)	(2)	Ok

Index	**Description of the issue**
(1)	Python files are not fully compliant with flake8 (pycodestyle, pyflakes, mccabe) standard
(2)	Not all release tags are SemVer compliant
(3)	Docs are not fully compliant with markdownlint standard
(4)	A CODE_OF_CONDUCT file is not present in the code repository
(5)	A CONTRIBUTING file is not present in the code repository
(6)	Found security weaknesses when performing SAST checks with bandit tool
(7)	No matching files found for language <CodeMeta> in repository searching by extensions or filenames No matching files found for language <Citation File Format> in repository searching by extensions or filenames
(8)	Software metadata failed to validate
(9)	JSON files are not fully compliant with jsonlint standard

Table 2: Evolution of the quality assessment criteria of the releases of *udocker*.

[23] See examples of badges and descriptions https://www.eosc-synergy.eu/eosc-synergy-100-software-quality-digital-badges-awarded/

Ongoing developments are being funded by the project “An interdisciplinary Digital Twin Engine for science” (interTwin) that has received funding from the European Union’s Horizon Europe Programme under Grant 101058386. Also, project "A Digital Twin for Geophysical Extremes" (DT-GEO) funded by Horizon Europe under the grant agreement No 101058129.

References

[1] H. Koziolek, Sustainability evaluation of software architectures: a systematic review, in: Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS, 2011, pp. 3–12.

[2] J. Axelsson, M. Skoglund, Quality assurance in software ecosystems: A systematic literature mapping and research agenda, Journal of Systems and Software 114 (2016) 69–81.

[3] I. Atoum, M. K. Baklizi, I. Alsmadi, A. A. Otoom, T. Alhersh, J. Ababneh, J. Almalki, S. M. Alshahrani, Challenges of software requirements quality assurance and validation: A systematic literature review, IEEE Access 9 (2021) 137613–137634.

[4] R. Vicente-Saez, C. Martinez-Fuentes, Open science now: A systematic literature review for an integrated definition, Journal of business research 88 (2018) 428–436.

[5] I. Global, Open badges specification (2023).
URL `https://www.imsglobal.org/spec/ob/v3p0`

[6] T. Galli, F. Chiclana, F. Siewe, Software product quality models, developments, trends, and evaluation, SN Computer Science 1 (05 2020). `doi:10.1007/s42979-020-00140-z`.

[7] ISO Central Secretary, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models: ISO/IEC 25010:2011 (2017).
URL `https://www.iso.org/standard/35733.html`

[8] C. Gacek, B. Arief, The many meanings of open source, IEEE Software 21 (1) (2004) 34–40. `doi:10.1109/MS.2004.1259206`.

[9] A. Adewumi, S. Misra, N. Omoregbe, A review of models for evaluating quality in open source software, IERI Procedia 4 (2013) 88–92, 2013 International Conference on Electronic Engineering and Computer Science (EECS 2013). `doi:https://doi.org/10.1016/j.ieri.2013.11.014`.
URL `https://www.sciencedirect.com/science/article/pii/S2212667813000178`

[10] P. Perera, R. Silva, I. Perera, Improve software quality through practicing devops, in: 2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer), 2017, pp. 1–6. `doi:10.1109/ICTER.2017.8257807`.

[11] L. König, A. Steffens, Towards a quality model for devops, Continuous Software Engineering Full-scale Software Engineering 37 (2018) 37–42.

[12] P. Orviz, A. Lopez Garcia, D. C. Duma, G. Donvito, M. David, J. Gomes, A set of common software quality assurance baseline criteria for research projects (2017). `doi:10.20350/DIGITALCSIC/12543`.
URL `https://digital.csic.es/handle/10261/160086`

[13] P. Orviz Fernández, M. David, J. Gomes, Joao Pina, S. Bernardo, I. Campos Plasencia, G. Moltó, M. Caballer, EOSC-synergy: A set of Common Service Quality Assurance Baseline Criteria for Research Projects (Jun. 2020). `doi:10.20350/DIGITALCSIC/12533`.
URL `https://digital.csic.es/handle/10261/214441`

[14] D. Salomoni, I. Campos, L. Gaido, J. M. De Lucas, P. Solagna, J. Gomes, L. Matyska, P. Fuhrman, M. Hardt, G. Donvito, L. Dutka, M. Plociennik, R. Barbera, I. Blanquer, A. Ceccanti, E. Cetinic, M. David, C. Duma, A. Lopez-Garcia, G. Molto, P. Orviz, Z. Sustr, M. Viljoen, F. Aguilar, L. Alves, M. Antonacci, L. A. Antonelli, S. Bagnasco, A. M. J. J. Bonvin, R. Bruno, Y. Chen, A. Costa, D. Davidovic, B. Ertl, M. Fargetta, S. Fiore, S. Gallozzi, Z. Kurkcuoglu, L. Lloret, J. Martins, A. Nuzzo, P. Nassisi, C. Palazzo, J. Pina, E. Sciacca, D. Spiga, M. Tangaro, M. Urbaniak, S. Vallero, B. Wegh, V. Zaccolo, F. Zambelli, T. Zok, INDIGO-DataCloud: a Platform to Facilitate Seamless Access to E-Infrastructures, Journal of Grid Computing 16 (3) (2018) 381–408. `doi:10.1007/s10723-018-9453-3`.
URL `http://link.springer.com/10.1007/s10723-018-9453-3`

[15] D. Cesini, A. Costantini, P. Fuhrmann, F. Aguilar, C. Duma, C. Ohmann, R. Lemrani, O. Keeble, S. Battaglia, V. Poireau, M. Viljoen, G. Donvito, The extreme-datacloud project: data management services for the next generation distributed e-infrastructures, in: 2018 Conference Grid, Cloud & High Performance Computing in Science (ROLCG), 2018, pp. 1–4. `doi:10.1109/ROLCG.2018.8572025`.

[16] A. Calatrava, H. Asorey, J. Astalos, A. Azevedo, F. Benincasa, I. Blanquer, M. Bobak, F. Brasileiro, L. Codó, L. del Cano, B. Esteban, M. Ferret, J. Handl, T. Kerzenmacher, V. Kozlov, A. Křenek, R. Martins,

M. Pavesio, A. J. Rubio-Montero, J. Sánchez-Ferrero, A survey of the european open science cloud services for expanding the capacity and capabilities of multidisciplinary scientific applications, Computer Science Review 49 (2023) 100571. doi:https://doi.org/10.1016/j.cosrev.2023.100571.
URL https://www.sciencedirect.com/science/article/pii/S1574013723000382

[17] M. David, M. Colom, D. Garijo, L. J. Castro, V. Louvet, E. Ronchieri, M. Torquati, L. del Caño, S. H. Leong, M. Van den Bossche, I. Campos, R. Di Cosmo, Task Force Sub Group 3 - Review of Software Quality Attributes and Characteristics (Aug. 2023). doi:10.5281/zenodo.8221384.

[18] The Open Source Definition (2022).
URL https://opensource.org/osd

[19] P. Orviz, A. Lopez Garcia, D. C. Duma, G. Donvito, M. David, J. Gomes, I. Campos, G. Moltó, V. Tykhonov, A set of common software quality assurance baseline criteria for research projects (2022). doi:10.20350/DIGITALCSIC/12543.
URL https://digital.csic.es/handle/10261/160086

[20] S. Bradner, Key words for use in RFCs to Indicate Requirement Levels, Tech. Rep. RFC2119, RFC Editor (Mar. 1997). doi:10.17487/rfc2119.
URL https://www.rfc-editor.org/info/rfc2119

[21] A. M. Smith, D. S. Katz, K. E. Niemeyer, Software citation principles, PeerJ Computer Science 2 (2016) e86.

[22] A. L. Mesquida, A. Mas, E. Amengual, J. A. Calvo-Manzano, It service management process improvement based on iso/iec 15504: A systematic review, Information and Software Technology 54 (3) (2012) 239–247.

[23] K. Morris, Infrastructure as code: managing servers in the cloud, " O'Reilly Media, Inc.", 2016.

[24] A. C. Miguel Caballer, German Molto, I. Blanquer, Infrastructure manager: A tosca-based orchestrator for the computing continuum, Journal of Grid Computing 21 (1) (Sep. 2023). doi:10.1007/s10723-023-09686-7.
URL https://link.springer.com/article/10.1007/s10723-023-09686-7

[25] A set of common software quality assurance baseline criteria for research projects (2023).
URL https://github.com/eosc-synergy/sqaaas-tooling

[26] J. Gomes, E. Bagnaschi, I. Campos, M. David, L. Alves, J. Martins, J. Pina, A. López-García, P. Orviz, Enabling rootless linux containers in multi-user environments: The udocker tool, Computer Physics Communications 232 (2018) 84–97. doi:https://doi.org/10.1016/j.cpc.2018.05.021.
URL https://www.sciencedirect.com/science/article/pii/S0010465518302042