Automation-Driven Quality Assurance

strategy for

Research Software

as the vehicle to consolidate the

European Open Science

Automatización del Análisis de Calidad del Software de Investigación

como vehículo para la consolidación de la

Ciencia en Abierto en Europa

Dissertation submitted by / Memoria presentada por

Pablo Orviz Fernández

to opt for the degree of Doctor of Science and Technology in the University of Cantabria / para optar al título de Doctor en Ciencia y Tecnología de la Universidad de Cantabria

June 2020 / Junio 2020

Supervised by / Dirigida por

Dra. Isabel Campos Plasencia
Dr. Álvaro Lopez García

Resumen global (en español)

De acuerdo con el enorme respaldo internacional recibido por parte de actores e instituciones de referencia dentro de la investigación científica, incluyendo organismos de financiación y agencias especializadas tales como la UNESCO, la Ciencia en Abierto se considera como el inminente salto cualitativo y revolucionario dentro del campo de la ciencia, decisivo para el impulso de la innovación y el intercambio de conocimiento científico. Dentro de las fronteras europeas, la transición hacia el nuevo paradigma de la Ciencia en Abierto se encuentra en pleno auge y desarrollo, tal y como indica la consecución del primer prototipo de un cloud europeo de Ciencia en Abierto, denominado European Open Science Cloud.

El software abierto, así como el acceso y los datos en abierto, es uno de los pilares fundamentales de la Ciencia en Abierto. De hecho, como resultado de la fuerte dependencia que la investigación moderna ha establecido con la computación de datos intensiva, el software surge como la pieza clave para conducir la innovación científica en el futuro y en un número de disciplinas cada vez mayor. Sin embargo, de acuerdo con el análisis de la literatura científica relacionada con la Ciencia en Abierto llevado a cabo en la redacción de la presente tesis, el software no siempre es considerado como ciudadano de primera clase en el ámbito científico, siendo en ocasiones desvirtuado en favor de otros productos resultantes del estudio científico, como es el caso de los datos resultantes de la investigación. Esta realidad es particularmente perceptible en el caso europeo, objeto de estudio de esta tesis, tal y como se puede extraer de los primeros pasos dados en la implementación del European Open Science Cloud.

Este ecosistema aspira a convertirse en el punto de acceso universal para todos los investigadores europeos, mediante la oferta de un catálogo completo de servicios útiles para llevar a cabo la investigación científica en diferentes campos, de acuerdo con los valores promulgados por la Ciencia en Abierto. Sin embargo, en el momento de la redacción del presente trabajo, los criterios para evaluar

la calidad y la madurez de los servicios ofertados no consideran el papel activo del software como elemento habilitador para el adecuado funcionamiento de dichos servicios. Como resultado, existe un asesoramiento y control inadecuado de los criterios mínimos de calidad, que en el peor de los casos, podría exponer al European Open Science Cloud ante la situación de suministrar, a través de su catálogo, servicios inestables que no cumplen con los requisitos y expectativas de los usuarios, ya sean científicos u otros actores dentro del ámbito de la investigación.

La presente tesis detalla un proceso automático para garantizar la calidad del software de investigación, tomando como referente la cultura DevOps, desde las fases iniciales del desarrollo de software hasta su despliegue final como servicio. Este proceso se propone como modelo para garantizar la calidad de los servicios del European Open Science Cloud, como paso previo a su integración en el catálogo. La viabilidad del proceso propuesto está avalada por su aplicación, y progresiva evolución, durante el transcurso de una serie de proyectos de desarrollo de software y servicios para e-Infrastructuras europeas, analizados a lo largo de este trabajo. Estas e-Infrastructuras disponen de una experiencia sólida, forjada a lo largo de los años, en el desarrollo, operación y soporte de servicios centrados en el usuario, y como tal, son un referente para el suministro de los servicios del European Open Science Cloud.

Principales conclusiones. Aun estando contextualizado en el marco europeo, y dentro del paradigma de la Ciencia en Abierto, las contribuciones y conclusiones extraídas a lo largo de este estudio son igualmente aplicables en un contexto científico más amplio o global. El principal objetivo es contribuir a realzar el valor del software como elemento habilitador de la reproducibilidad en la ciencia y acelerador de la innovación en la investigación moderna. Para ello, en los sucesivos capítulos se presenta un manifiesto que fomenta el compromiso con una cultura de calidad en la producción de software científico o de investigación.

El paradigma de la Ciencia en Abierto ofrece el contexto perfecto para el

establecimiento de dicha cultura de calidad para el software científico, y como tal, las principales contribuciones del presente trabajo están orientadas a abordar los retos e inflexiones puestas de manifiesto en la actual implementación de la Ciencia en Abierto en Europa, especialmente en relación a la calidad de los servicios ofrecidos a la comunidad investigadora. Así, la usabilidad del servicio es primordial para el éxito del European Open Science Cloud, y ésta depende en gran medida de la calidad del servicio, no sólo en términos de estabilidad y fiabilidad, sino también en relación a la idoneidad de sus capacidades funcionales, necesarias para satisfacer las expectativas de los investigadores.

El presente trabajo concluye que la ***aplicación práctica de una cultura de calidad del software de investigación, gobernada por el manifiesto anteriormente mencionado y operada por medios automáticos, tiene un fuerte impacto en la usabilidad del futuro servicio de investigación***. Como demostración, la segunda parte de esta tesis describe los resultados empíricos obtenidos en relación a la puesta en práctica de la metodología y criterios de calidad propuestos en proyectos de desarrollo de software y servicios avanzados para e-Infrastructuras de investigación europeas. Estas e-Infrastructuras llevan proporcionando desde hace más de una década recursos de computación intensivos y servicios de investigación avanzados a una plétora de comunidades científicas y, por tanto, son un modelo fiable para la implementación del European Open Science Cloud.

Contribuciones y resultados obtenidos.

- La ***definición de los criterios mínimos para asegurar la calidad del software de investigación a lo largo de su ciclo de vida***, incluyendo prácticas para la producción y análisis del código fuente, testado, mantenimiento y usabilidad. El manifiesto resultante es público y está abierto a colaboración externa, con objeto de establecer una vía sostenible para la generación y transferencia de conocimiento colectivo, que sirva como referencia en desarrollos de software científico prospectivos.

En el contexto del European Open Science Cloud, este documento sirve además como base para la estimación de la calidad del software en los servicios ofrecidos. Así, el software dispondrá de una serie de principios o criterios de calidad, actualmente inexistentes, que a día de hoy sólo son considerados para los datos de investigación dentro de la hoja de ruta establecida por la Unión Europea.

- El ***diseño y puesta en práctica de un proceso automático que implementa los criterios de calidad definidos en el manifiesto, apoyándose en la cultura DevOps, para desarrollar software con miras al futuro rendimiento operacional en las e-Infrastructuras europeas que contribuyen al European Open Science Cloud***.

  El principal resultado de este proceso es la composición de *tuberías de ćodigo* o *code pipelines*, que permiten implementar y ejecutar los criterios de calidad para cada cambio realizado en el código fuente. Para facilitar esta tarea, se ha desarrollado una librería que implementa las funcionalidades comúnmente requeridas en entornos de integración continua. Las *code pipelines* permiten un alto grado de portabilidad, de manera que pueden ser reutilizadas en diversos entornos, con lo que las prácticas de calidad allí implementadas persisten durante la vida útil del producto software, perdurando más allá de la duración de los proyectos de investigación en los que se han desarrollado.

- La ***demostración de la aplicabilidad del mencionado proceso de calidad no sólo en los casos de software crítico para la operación de las e-Infrastructuras, habitualmente desarrollado por expertos en ingenería del software, sino igualmente adecuado para software desarrollado por científicos computacionales, de origen multidisciplinar***. Muestra de ello es la sucesiva implementación de soluciones DevOps para el desarrollo y distribución de aplicaciones científicas en los proyectos del Programa Marco Horizon 2020, INDIGO-DataCloud y DEEP-Hybrid-DataCloud. En este último proyecto se consolidó una

solución particularmente avanzada para la disponibilidad continua de aplicaciones de *deep learning* a través del catálogo del proyecto.

- La ***modernización del proceso de provisión de software dentro de la federación European Grid Infrastructure***, e-Infrastructura clave en la implementación del European Open Science Cloud, con el objetivo de optimizar la fiabilidad y tiempo de entrega del software suministrado a través de sus dos distribuciones oficiales. La revisión del proceso incluye la gradual adopción de prácticas de automatización, concluyendo en la implementación de un proceso DevOps para llevar a cabo la continua validación del software antes de ser distribuido por los canales oficiales.

- El ***diseño de un servicio, denominado Software Quality Assurance as a Service y desarrollado como parte de las actividades del proyecto EOSC-Synergy, para promover y sustentar la cultura de calidad en el software científico dentro del ecosistema europeo de la Ciencia en Abierto***. Por un lado, este servicio se apoya en el anteriormente mencionado manifiesto para evaluar de forma automática la calidad del software científico, y reconociendo sus cualidades mediante la emisión de distintivos o sellos digitales, como primer paso hacia la implantación de un sello de calidad oficial a nivel europeo. Por otro lado, el servicio ofrece la capacidad de componer *code pipelines* personalizadas, con funcionalidades relacionadas con el desarrollo y distribución de software, de acuerdo con los propósitos y aspiraciones del científico computacional.

Desarrollos futuros. De acuerdo con las principales contribuciones recién expuestas, dos hitos fundamentales se preveen en un futuro inmediato.

- La ***definición de un manifiesto específico para la calidad de los servicios***, siguiendo el modelo establecido por los criterios de la calidad del software. Aunque están estrechamente relacionados, los factores de calidad de los servicios están íntegramente relacionados con su usabilidad, y por tanto, difieren en cuanto a la perspectiva de aplicación. Así, para

evaluar la calidad de un servicio se debe analizar como una *caja negra* en la que sólo su comportamiento, en términos de idoneidad funcional, es relevante. Otros aspectos operacionales relacionados con la infrastructura, como la seguridad o condiciones de uso, deberán ser igualmente considerados. Debido a que el European Open Science Cloud solamente considera servicios, y no explícitamente software, la redacción de este documento es especialmente significativa.

- La ***instauración de una certificación oficial de calidad de servicios de investigación a nivel europeo***. Uno de los informes publicados en la fase de consulta de la implementación del cloud europeo de Ciencia en Abierto, *Prompting an EOSC in practice*, recomendaba el establecimiento de un sistema de certificación para "promover la sostenibilidad a largo plazo de la operación del European Open Science Cloud y dar crédito a desarrollos de software innovativos". La implementación de la anteriormente mencionada solución *Software Quality Assurance as a Service* proveerá de un prototipo para la emisión de sellos de calidad para servicios de computación, de acuerdo con los criterios establecidos en el manifiesto de calidad del software. La adopción de esta solución deberá ser considerada no sólo como medio de certificación y reconocimiento de la calidad del software, sino como herramienta de evaluación, formación y regulación tanto para el mantenimiento del software existente como para el futuro desarrollo de software de investigación. De esta manera, el software ocupará su lugar legítimo dentro de la investigación moderna.

Abstract

Given the strong endorsement by the research stakeholders worldwide, including national and international funding bodies and specialized agencies such as UNESCO, the Open Science movement is considered as the next quantum leap in science, decisive for accelerating innovation and knowledge sharing. In the particular case of the European Union, the transition towards Open Science has already started, undertaking the preliminary steps towards the establishment of an European Open Science Cloud.

Open source software, in addition to open access and open data, has been identified as one of the main pillars of Open Science. Indeed, as modern research is requiring data-intensive computing, software is deemed as the cornerstone to drive the innovation in an increasing number of scientific disciplines. However, according to the analysis of Open Science-related literature conducted in this thesis, software is commonly regarded as a second class citizen, not being equally treated as the other research objects, such as data. This fact is particularly noticeable in the European case, subject of study of the present work, when examining the initial steps taken in the implementation of the European Open Science Cloud.

This ecosystem aims at constituting an universal entry point for all the European researchers, offering them a catalogue of research-enabling services to do science according to the Open Science values. However, at the time of writing, these services are not being accurately assessed in terms of quality and maturity, and the role of the underlying software, that enables the services, has been disregarded to a great extent. As a result, there is an inadequate guidance and control of the minimum quality standards, which can pose the risk to the European Open Science Cloud of delivering unreliable services that not meet the researchers' expectations.

This thesis presents an automation-driven Software Quality Assurance process, built upon a DevOps culture, to govern the whole life-cycle of software

production, from the early development stages to the final delivery as services, thus serving as a model for the quality assessment and integration of services in the European Open Science Cloud. This process is established on a strong foundation, evolving throughout the years of guiding the software delivery of several research e-Infrastructure development projects. The European e-Infrastructures have a solid background, forged over the years, on the development, operation and support of user-centric services, and accordingly, can be regarded as a benchmark for the delivery of services within the ecosystem of the European Open Science Cloud.

As part of the outcomes of this thesis, concrete guidance is provided to drive the development life-cycle of quality research software. Furthermore, a quality assessment tool is presented, which is intended to contribute to address the aforementioned challenges with regards to the viability of the service onboarding in the European Open Science Cloud.

Although the work is contextualized under the European scope, the outcomes and conclusions presented are equally applicable in a wider or global context. The main goal is to contribute to enhance the value of software in the scientific community as a key research object for the realization of the Open Science. Research software is the enabler of reproducibility in science, as well as an innovation accelerator in modern research. Only by ensuring the quality of research software, the aforementioned goals can be accomplished.

Acknowledgements

Needless to say that one single individual cannot embark alone on the undertaking being described throughout the following chapters. The practical implementation of the most innovative ideas requires being surrounded by skilled partners that deeply believe in achieving a common goal, and in my case, I had the luck of having highly valued travel companions over the years and research projects. I am particularly thankful to my Portuguese colleagues at Laboratório de Instrumentação e Física Experimental de Partículas (LIP) –especially Joao, Mario, Jorge and Samuel–, and Cristina from the Istituto Nazionale di Fisica Nucleare (INFN-CNAF). Moreover, since groundbreaking ideas always came from most bright minds, I would like to highlight the pivotal role that my friend and next-door colleague (and thesis co-director) Álvaro had in the work presented through constant advice and vision. Last but not least, very grateful to the thesis advisors, Isabel and Francisco, always there when needed, and the rest of my department colleagues at Instituto de Física de Cantabria (IFCA). I should not forget COVID-19 lockdown, which definitely made me sit up and write this dissertation.

Contents

List of Figures

List of Tables

Thesis Statement

1

Thesis Statement

Contextualization and Objectives

Software is an indispensable tool for conducting research. The term *research software* is used throughout this thesis to refer to the type of software that produces, analyses and visualizes the data in a scientific study, or is accessory to it. Hence, research software and data are mutually inclusive, but, whilst much of the global attention has been put on data, the former has been traditionally overshadowed.

The advent of the Open Science movement, increasingly considered as the future of science, has raised great expectation not only within the research ecosystem, but also in the society, as it fosters the democratization of the scientific knowledge, making it readily accessible for all the individuals and countries.

In a research context, one of the most prominent goals of Open Science is to address the currently existing reproducibility crisis of the scientific outputs, and for that purpose, a major cultural shift is required within the scientific community.

According to the working hypothesis of this study, the quality of the software plays a strategic role as the legitimate enabler of the principles underpinned by the Open Science movement. To this end, the scenario of this study is placed in the context of the first stages of the Open Science implementation within the European Union. Here, the predominant effort has been put on delivering horizontal and thematic services, through the European Open Science Cloud (EOSC), that allow to conduct research and data management according to standards akin to Open Science principles, such as the FAIR principles.

However, as elaborated in subsequent chapters, the EOSC implementation is yet again disregarding the role of software in delivering those services, without providing adequate guidance and regulation. Accordingly, the main ***objective of this thesis is to demonstrate how the quality of the software contributes to meet the challenges that emanate from the implementation of the Open Science values in the services offered through the EOSC***. Hence, the present work places great emphasis on building a Software Quality Assurance (SQA) culture for the development of research software, providing empirical evidence throughout the chapters from the second part.

Document structure

The content is broken down in two main blocks, structured as follows:

- The first three chapters are clustered around the "Software Quality, Open Science and European Research Ecosystem" block, which provides a proper contextualization of the work and introduces the essential concepts and tools that are pivotal in order to discuss the main outcomes that are laid out in the second block "A Story of Three Acts".

  In particular, Chapter 2 "The Role of Software in Open Science" exhibits the current status and challenges that the Open Science paradigm seeks

to address, putting specific emphasis on the strategic role of the software.

In Chapter 3 "Building a Culture of Software Quality", the foundations for establishing a culture of software quality are set by scouting the desirable quality characteristics, as well as the state-of-the-art methodologies that will provide the tools for the practical implementation of such a culture in the second block.

The scope of the discussion will then be framed into the Open Science implementation in the European landscape, as part of Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud". A comprehensive analysis of the first steps taken for the realisation of the EOSC is herein presented, outlining the currently existing strategic and technological gaps that are preventing the successful delivery of the Open Science values.

- The second block, "A Story of Three Acts", presents the major outcomes of this study as a logical sequence of acts that cover the definition, implementation and dissemination of the aforementioned SQA culture in the European e-Infrastructures, and thus, in the EOSC. Consequently:

  1. Act I "Laying out the Groundwork: The Definition of a Baseline for Software Quality Assurance" lays out the groundwork for the subsequent phases through the formulation of a comprehensive criteria, coined as *SQA baseline*, that identifies the flagship topics to be addressed while driving the development and maintenance of the research software.
  2. Act II "Where Theory Meets Praxis: the Implementation Process" presents real use cases of past and ongoing e-Infrastructure projects where the herein proposed SQA culture has been successfully implemented. The chapters of this act cover the DevOps approaches to cope with the the development, deployment and operation phases, which are relevant to serve as a model to improve the delivery of services in the EOSC.

3. Act III "Mapping out the Future: Universalize and Sustain a Culture of Quality Research Software" introduces the *SQA-as-a-service (SQAaaS)* solution, which harnesses the insight and knowledge accumulated over the course of the previous projects, putting them within reach of the global scientific community. The SQAaaS provides a means to accurately assess, and award, the quality of the EOSC software and services.

The last blocks of the thesis are reserved for the formulation of the "Conclusions", which captures the main contributions of this thesis and discusses the future directions and possible improvements of the outcomes herein obtained, and the "Appendices" that collects the relevant appendices of this thesis.

Software Quality, Open Science and European Research Ecosystem

"Information wants to be free"
Stewart Brand

2

Present and Future of Software in Research: the Role of Software in Open Science

Open Science has the simple, and yet ambitious, objective of making all the present and future scientific advances within global reach, thus not only being open to the research community but also to the society at large. The democratization of the scientific knowledge will build on collaboration and sharing values to unlock the full potential of innovation in research, as all this knowledge will be readily available by all the stakeholders in the research process, extending the dissemination levels to citizens worldwide.

In this chapter, Open Science will be presented as the qualitative step ahead for research advancement. However, the Open Science realisation requires both a cultural shift in the scientific community, to empower the creation of collaborative knowledge and subsequent transparent dissemination, and the proper identification of the pillars that will sustain the movement. On this latter point, the role of the software is emphasized as the principal enabler of reproducible research, key goal to achieve such quality leap.

2.1 Ode to Open Science: addressing the reproducibility crisis

Open Science movement stands for the global cooperation and dissemination of scientific knowledge, across all disciplines and accessible by all the individuals and stakeholders, reaching the long tail of research. No country, institution or scientific discipline shall be disfavored in the exploitation of Open Science, regardless of the geographical location, income status or regional policies. According to the UNESCO, "Open Science could be a game changer for achieving the United Nations Sustainable Development Goals" [1].

The current landscape is promising. New digital technologies pave the path for the practical implementation of the Open Science culture by removing the barriers for collaborative work from the very early stages of research process. International bodies such as the Organisation for Economic Co-operation and Development (OECD) drafted guidelines [2] that set the standards to be followed for establishing Open Science. Other research community organisations such as the Committee on Data for Science and Technology (CODATA) or the Research Data Alliance (RDA) advocate for international collaboration to advance Open Science and to accelerate the transition to immediate accessibility and usability of data for all areas of research. The European Commission (EC) is pioneering at the governmental level the implementation of the Open Science paradigm since the publication of the *Open Innovation, Open Science and Open to the World* [3]

report, which constituted the starting point for materialising the open access to scientific data and the development of an European Open Science Cloud (EOSC).

The implementation of Open Science shall guarantee the promotion of openness, reproducibility and transparency values in scientific publications. But this implies educating the scientific community, in an attempt to drive researchers' behaviour towards the adoption of these values in daily practice, through the adjustment of the current incentive structures. Initiatives such as the Transparency and Openness Promotion (TOP) guidelines [4] promote the practice of Open Science values in scientific journals through the definition of concrete actions to change rewarding mechanisms in research towards open practices. Hence, the immediate outcome of the adoption of Open Science are both the extended capacity to reproduce scientific results, and subsequently, the building of trust in science by citizens.

2.1.1 The contextualization of the Open Science term

Since its inception, the Open Science term has generated debate about the scope of application among the diverse stakeholders involved in the research process, i.e. scientists, research infrastructure developers and operators, policy makers or citizens. It is an overarching concept that cuts across several domains such as the access and creation of the research knowledge, more aligned with the interests of the two former groups, or the outreach capacity of science to the general public, more in line with the expectations of politicians and society.

In this regard, Fecher and Friesike [5] identify five different schools of thought in Open Science, in which the aforementioned stakeholders are grouped according to their requirements in regards to the accessibility –equal, public, peer recognized– and creation –collaborative effort, tools and services– of scientific knowledge. A more precise attempt of providing a formal definition for Open Science was carried out by Vicente-Sáez and Martinez-Fuentes [6] through a systematic analysis based on the presence of the term and associated predi-

cates across interdisciplinary literature. The definition "Open Science is the transparent and accessible knowledge that is shared and developed through collaborative networks" includes keywords commonly present when describing the Open Science phenomenon.

The fact that Open Science concept is diversely interpreted reveals the popularity and the growing interest of the involved stakeholders in the realisation of a paradigm in science that makes all the publicly funded research knowledge globally accessible. Each group provides a different vision that reflects the existing gaps in their approach to science. Open Science evokes numerous concepts and therefore entails a broad and complex set of topics, some yet to be identified.

The unreleased *UNESCO Recommendation on Open Science* [7] is expected to build a global consensus on Open Science's overarching principles and values, through global and inclusive dialogue with all the member states, with the aim of reducing the science gaps "between the haves and have-nots". Until then, the definitions and approaches herein discussed, along with the enumeration of the pillars in the next section, provide a realistic approximation to the Open Science movement.

2.1.2 The Open Science pillars

Just as the Open Science definition is subject to different interpretations, there is no universal agreement about the principal pillars that sustain this paradigm. From three [8], through four [9] to six [10], the common understanding, as illustrated in Figure 2.1 is that Open Access, Open Data and Open Source are the fundamental prerequisites for the Open Science realization.

Open Access

Open Access is a well-known publishing model that entails general free-of-charge accessibility to digital scientific literature. Traditionally, there were three main ways for scientists to deliver publications in Open Access mode, either via *pre-*

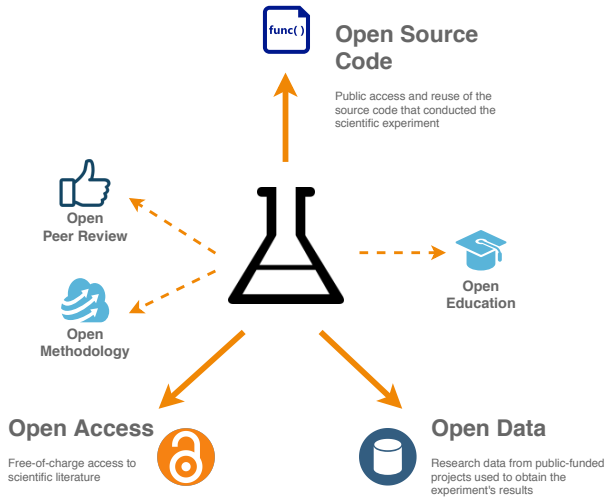

Figure 2.1: The three main pillars of Open Science

prints –articles before the formal peer review has been performed within the scientific journal's acceptance process– or *post-prints* –journal peer-reviewed articles that can be published after an embargo period– in online repositories, and paying to subscription-based research journals the production costs upfront as a way to avoid the establishment of access barriers to research advancements.

The advent of the Open Access journals provided a means to publish free-of-charge and peer-reviewed scientific content in Open Access. The widespread acceptance of this type of journals is illustrated by the metrics provided by the Directory of Open Access Journals (DOAJ) [11], which, at the time of writing, records more than 14K journals for a total of 4.7M articles. Furthermore, studies [12] have shown that the funding mechanism is irrelevant when considering the quality of a scientific journal, and as a result, the Open Access journals are closely approximating the same scientific impact and quality as the subscription journals. And this fact represents a major push to Open Science implementation.

Open Data

Open Data implies the public access and utilisation of an experiment's data –both raw (or primary) and processed (secondary, tertiary)–, relevant to the pursuit of the research value. Consequently, the extent of data should enable the execution of the same analysis conducted during the course of such experiment. Such lack of data availability is commonplace in scientific publications, thus hindering the solidness and, most importantly, the ***reproducibility of scientific results***.

This is specially controversial in the research conducted using public funds, where the valuable data is traditionally considered as intellectual property [8] of the researchers. Data outlives the scientific paper [10], and should not be hidden, rationed or *hijacked* by the researcher in order to exclusively extract the knowledge out of them, and rather, be openly available to other researchers. This is the only way that fast innovation, promoted by the Open Science movement, can be achieved. Needless to say that there are specific cases, usually involving ethical concerns, where data protection is a requirement and cannot be delivered publicly.

In Europe, the EC's Horizon 2020 Programme has set the goal to remove the barriers for accessing scientific outcomes, both in terms of publications and research data. For the latter, the Open Research Data (ORD) pilot [13] has been established to provide guidance in order to balance openness and protection of data in underlying publications.

Open Source

Open Source refers to the public access and reuse of the source code, which is the representation of the software in plain text using a programming language. The term was originally coined as a way to diverge from the philosophical implications of the *free software* term in order to promote business adoption. Eventually both terms co-existed over time, giving rise to the Free/Libre and Open

Source Software (FLOSS), which appears as a neutral term [1] that highlights their commonalities.

As the rest of Open Science pillars, open source concept was empowered by the digital transformation that occurred with the advent of the Internet technology. It enabled a new methodology underpinned by the collaborative development of software. To this end, the open source software is required to be delivered under a licensing agreement that not only makes the software readable, but also relinquished all creative or financial control over the code [2]. This methodology was firstly applied in the development of the Linux kernel project, and later on universalized by the code sharing platforms, such as Sourceforge [15] and GitHub [16].

In accordance with the "Thesis Statement" and the outcomes that will be presented in the next chapters, ***Open Source is a key enabler of the Open Science movement***, not only in terms of openness, intellectual property or accessibility in general –as it is commonly and uniquely found in the Open Science-related literature–, but also ***within a broader and inclusive context of software quality***.

2.1.3 The path to reproducibility in Science

Reproducible research is a popular topic and a particular aim in the current research scene, not only because of the recent focus on the reproducibility crisis, but also as a way to enhance modern science and the scholarly process. When talking about Open Science, and how this movement is gaining momentum as a solution to the reproducibility crisis in global research, most of the scientific

[1] See `https://www.gnu.org/philosophy/floss-and-foss.html` for further clarification about FLOSS and Free and Open Source Software (FOSS) terms.

[2] The open source models were represented by one of its founders, Eric S. Raymond, in his essay *The Cathedral and the Bazaar* [14]. While the cathedral model supported a closed –or non-open– software development approach, the bazaar model stands for source code freely shared and collaboratively developed by programmers worldwide, exploited through the potential of Internet. Through the "given enough eyeballs, all bugs are shallow" statement, Raymond expressed the fundamental benefit of open source over the traditional closed approach.

literature and research policies remark the need of a data-driven process. It is widely accepted that, even the Open Science term suggest otherwise, *open* is not the only element in the equation [17], but instead it needs to be supplemented by additional research practices during the data analysis process.

Debate in the scientific community on data sharing

The scientific literature from the early days of the Open Science implementation reflected some concerns about the upcoming shift to open data. In order to be valuable, open research data is required to be curated before being distributed and used by others, and thus, the sceptics argued about the process and cost of carrying out such process [18]. More recent studies gave light to data protection issues, presenting the risks of being plagiarized or even jeopardizing a collaboration as a result of discrepancies on adhering to Open Science principles [19].

However, the upsides of the new paradigm were also broadly acknowledged, and solutions to the identified risks started to emerge [20]. It is a fact that open publications have a higher number of citations, as not only the paper itself is cited, but also the other research objects such as data and software. Persistent Identifiers (PIDs) enable the citation of the research objects, and thus, highly contribute to the reproducibility of scientific results since the *paper-data-code* triangle is uniquely identified. Other risks such as plagiarism can be dissuaded by the use of time-stamped data, data deposition plans or temporary embargo periods for the most sensitive data.

In short, data sharing requires from a previous, complex process to produce and curate data in order to profit from the demonstrated benefits of an open data approach to science. Good practices are then needed to build a data culture within research that enables the Open Science implementation.

Establishing a data culture through the FAIR principles

Adopting good practices early in the data management process is key to enable the reproducibility and reusability of research data. In this regard, the unquestionable reference are the Findability, Accessibility, Interoperability and Re-usable (FAIR) principles formulated by Wilkinson et al. [21]. These provide a measurable way to attain good data management and stewardship, both for data producers and publishers, via a set of requirements categorized under four foundational principles: Findability, Accessibility, Interoperability and Reusability.

The FAIR principles enable an accurate use of metadata, treated equally as the data themselves. Considerations such as having PIDs both for data and metadata, licensing or the use of standarized ways to create or access the data, are part of the FAIR practices.

Since its inception, the FAIR data principles have been extensively embraced by scientific initiatives worldwide, but with special emphasis in the design of the Open Science implementation in Europe, as it will be showed later on in Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud".

What about Software?

According to what it has been stated so far, much of the global research focus has been put on data, overshadowing the role of the software in the Open Science movement. FAIR principles have been adopted by research institutions and even at the policy level, such as in the European Union (EU), as the reference for guiding data management and stewardship processes.

Software has not received the same treatment at the institutional level. Unlike the data, there is ***no research regulation or policy that defines the principles that should guide the development and maintenance of research software***. As a result, software has been already considered as the forgotten pillar of Open Science [22].

Box 1.1: EC expert group on FAIR data

The consultation report on FAIR data implementation [23], requested to data experts by the EC, states that "(FAIR principles) do not just apply to data but to other digital objects", revealing later on those objects as "metadata, identifiers, software and Data Management Plans (DMPs)".

In terms of reproducibility, software is certainly a key player. Chen et al. [17] advocates that the path to reusability and reproducibility does not consist exclusively on data issues, but they also go hand in hand with "software, workflows and explanations". Consolidated reproducibility guidelines [24] extensively promote the use of software engineering practices, such as the use of version control systems or verification and validation processes. Other guidelines, such as Goodman et al.'s [25], also underline the fact that "publishing the source code and its version history is crucial to enhance transparency and reproducibility".

Box 1.2: Software not part of the EC Open Science ambitions

The EC identified a set of recommendations for the implementation of Open Science in the European research ecosystem, collected under the Open Science Policy Platform (OSPP) recommendations [26] document, released in April 2018. Contrary to most research literature on Open Science, none of the formulated requirements in this document considers open source –or software– as a prioritized issue to be promoted towards the implementation of Open Science in Europe. This fact was subsequently acknowledged by the League of European Research Universities (LERU) in an advice paper [27], highlighting that "the European Commission has identified eight component parts of Open Science, but universities may feel that there are additional areas that should be catered for. Copyright regimes allied to Open Science principles, infrastructure development, sustainable research software, open education, and artificial intelligence are examples of areas which are not explicitly treated in the Commission's vision".

2.2 Research software in Open Science

Software is present in a lot of disparate fields in modern research, ranging from sciences to engineering to humanities. It is being considered as an indispensable tool for doing research, either as a value object by itself or to conform the groundwork that provides the scientific result [28]. The term *research software* identifies with the kind of software that is used for generating, processing, analysing and visualizing data in a scientific study. Accordingly, research software is both *diverse*, it might be targeted to address a complex simulation in a high-performance computer or to solve a simple problem, and *specialized*, dwelling within a very specific application domain. As a consequence, it is commonly developed and maintained by the scientific community doing the research [29].

Several studies have confirmed the growing presence of software in reference journals such as Nature, where an examination of the published papers over a 3-month period showed that 80% of them had mentioned software and/or referred to research software tools [30]. Therefore, according to surveys [31, 32], more and more scientists are relying on software tools to carry out their work, both in universities or other academic-related organisations, to such an extent that their research would be hard or even impossible to be conducted without software assistance.

2.2.1 Common problems in research software development

Despite the widespread adoption in academic environments, software is still not treated as a value object in modern research. As a result, a series of problems, associated with the involved stakeholders –mainly scientists, universities and funders–, characterize and hinder the application of good practices in research software development [33]. Figure 2.2 summarizes the source of those problems that are both inherent and accidental to the academic research development. The ***inherent problems are the most relevant to this thesis*** as they are

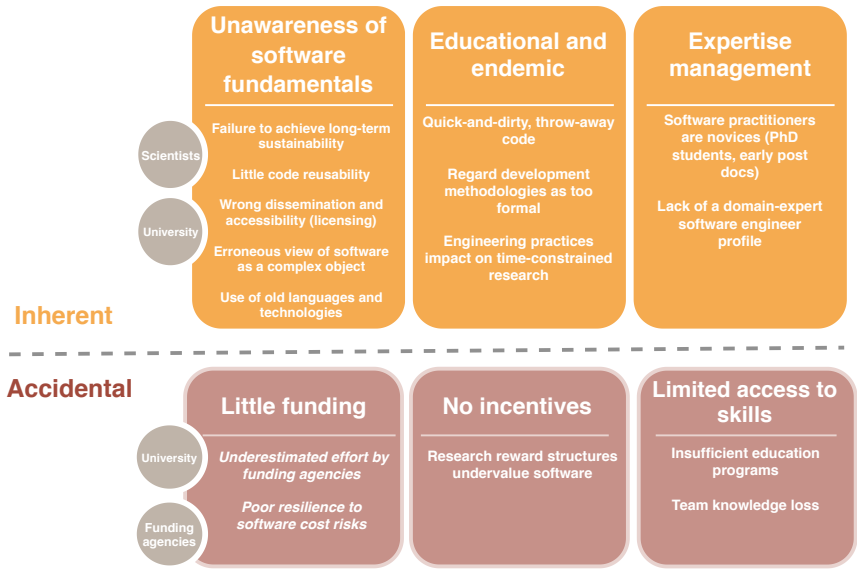

Figure 2.2: Common problems of software developed in academia

resultant from bad practices that, in some cases, do not even provide any profit to the researchers, and, in most cases, inevitably lead to future burden on software development and maintenance tasks. In software engineering, the term *technical debt* expresses this issue, as *quick and dirty* code incur debt to obtain rapid benefit –a program that does its job–, but such debt will be paid by future code maintainers and/or users.

Inherent problems

Software was traditionally developed in an ad hoc manner, to ***serve promptly the specific needs of a given research activity, without worrying about the long-term sustainability of the solution***. Consequently, in such cases, no design principle or best practice was followed, only individual, and usually random, decisions done by the developers at the time of coding. In many oc-

casions, those software projects become popular in the scientific domain have prevailed for a longer time than it was first planned. It is then when development and maintenance problems might arise.

As it will be described in the second part of this thesis, the experience of managing software development in academic research has shown that the ***lack of awareness of the fundamentals of software engineering*** theory is a recurrent issue. But even if scientists are familiar with software engineering practices, they are tempted to develop *quick and dirty* code in order to come up with a solution in the short term. This may be a practical solution for very specific and rare situations, such as when developing *throwaway code* to serve its purpose during a very short period of time and never reuse it anymore. However, when it comes to software meant to be reused, it is indeed a deceptive practice since the cost of maintenance grows with the increase in the program size. *Software reusability* is then a key concept in order to illustrate the virtues of the quality software. Specially in small teams, there are recurring situations where the developer of a program or application stops maintaining it at a given point in time,

Underestimating the value of quality software, or putting it differently, the bad consequences that unstructured code might bring along, ***is a educational and endemic problem in academia***. To illustrate this, one could think about a project coordinator or senior researcher that sets the policy of refusing the application of well-known software engineering conventions, so that any software solution that derives from within the project or department work will be lacking of any type of quality control. Such opposition is not that uncommon and may arise from meeting time constraints or, in the specifics of developing cutting edge solutions, to the fact that the research product might not be tangible in the event of investigations not producing any outcome, so why spend time on software quality then? Mentoring young researchers in software engineering good-enough practices would set the path to confront this problem in academia.

In this regard, a widespread fault in the academic system points to the

general ***management of the software-related expertise within research groups***. Probably as a result of not treating software as a first-class object, universities and research organisations are currently lacking from a profile that both understands the research processes and has the software engineering background and skills in order to carry out, in a more professional manner, the development of research software. In order to fill this gap, in recent years there have been an effort in order to define the career paths of such a profile, naming it as Research Software Engineer (RSE) [34], who differs from a technology expert in that the RSE is, much like the scientist, an expert in the particular scientific domain. RSE-type jobs are increasingly emerging across research organizations [35] and are expected to change the current organizational structures for research software development in academia.

Accidental problems

More than usual, the underlying problems cannot be ascribed purely to the researchers themselves, but instead to the public funding agencies. Limited funding is a barrier to carry out the required actions when developing quality software, usually in terms of required personnel. Furthermore, this leads to serious difficulties to overcome the occurrence of risks associated to software development [36].

Accidental issues encompass also the lack of proper reward systems, which should favor source code contributions in addition to the current structure based on the publication of research articles [37]. In order for this code contributions to be reused, funding agencies should be required to promote educational initiatives to improve the software engineering skills of computational researchers. A recent survey [32] reflects that self-taught computational post-doctoral scientists still represent more than a half (54%) in the US, where the 95% of respondents commonly use research software and 63% state they could not do their research without research software.

2.2.2 Reproducibility in Software

The advent of the Open Science paradigm is a great opportunity to empower the role of research software. Reproducibility is a fundamental principle of science, and thus, the underpinning goal of Open Science. However, the nature of software challenges the ideal of achieving full reproducibility, even before considering the availability of inputs. Software might not be deterministic for all the possible use cases, indeed challenges will arise when trying to reproduce the conclusions reached previously by other researcher, but as it will be shown in this section, a set of decisive factors are prerequisites for attaining reproducibility as far as research software is concerned.

The three R's

Assuming the availability of data, the reproducibility of the software used as part of the original experiment is often associated with other related terms that enable or require it.

In the first place, *replicability* implies tackling the very same steps followed in the original experiment, aiming at obtaining the same results. To this end, the same execution environment –such as hardware, compiler, libraries– shall be required. Replicability is also known as repeatability [38], which is dependant on the deterministic nature of the software.

Instead, *reproducibility* does not really require that the same environment is mirrored, but that the same conclusions of the original experiment are reached, even though this implies some changes in the process followed. Obviously, the differences shall not be significant for the final result and both processes shall be equivalent. Running the original version of the source code in a different platform or refactoring the code to implement the same algorithm are different forms of reproducibility [39]. Accordingly, reproducibility is the cornerstone of science –or the "bread and butter" of doing science [38]– as it finds alternative ways to corroborate the findings obtained during a previous scientific experiment. Successful reproducibility points to an extended reliability of both such

findings and the process followed to achieve them.

Lastly, *reusability* builds on the capacity of the software to be encapsulated and reused for additional purposes other than the original intention. Reuse often transcends the boundary of the projects where the software was first designed.

Consequently, the three R's [40] are tightly coupled: replicability or repeatability is the minimum level of scientific integrity, except in the case that software systems are non-deterministic by design. Reproducibility is the catalyst of such integrity or scientific rigor through the corroboration of the reliability of the process followed to obtain the research value. Reusability is, on the other hand, an engaging result of achieving reproducibility, and a key objective when enhancing the quality of the software.

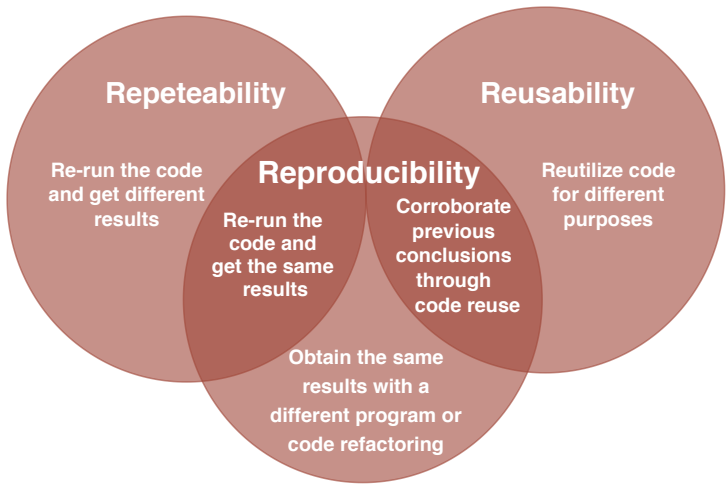

Figure 2.3: Addition of reusability in the Easterbrook's comparison of repeteability and reproducibility. While reproducibility alone was the most relevant outcome according to Easterbrook's analysis, the emergence of reusability changes the game since reproducibility is best promoted when both are used in conjunction, i.e., by ***corroborating previous conclusions through code reuse***.

The reproducibility and reusability marriage

The differences between replicability and reproducibility in scientific computing have been clearly spotted by Easterbrook [38]. Replicability by itself is relevant from an engineering perspective as it seeks the deterministic behaviour of the software through testing methodologies. As such, re-running the software and not getting the same results is a clear sign of poor code. On the other hand, replicability is not that relevant from a research perspective, as it implies repeating the experiment following the original approach, and thus, it does not yield new scientific insights.

Figure 2.3 applies Easterbrook's method to compare reproducibility and reusability. As it can be seen, there are a few differences with the previous case. First, the three possible outcomes are positive so it is safe to say that both are complementary. And secondly, the main outcome is placed in the intersection, not at the reproducibility side as it happened in previous comparison. Consequently, ***achieving reproducibility through reuse is the most effective way to do science***, as reproducibility without reuse would imply the higher costs of producing new algorithms, and reuse by itself does not provide any benefit when trying to corroborate the conclusions reached by a previous scientific study.

Software reusability is a key objective in software engineering, which is either the unique purpose of producing the software, such as the case of libraries, or an intended outcome when developing an end-to-end solution. Software engineering literature [41] pictures both processes as *construction for reuse* and *construction with reuse*, respectively. And both require high standards of Software Quality Assurance (SQA), specifically in coding and testing.

The scientific computing literature also states that reusability is a necessary prerequisite for reproducibility [42], and viceversa [40]. The bottom line is that both concepts can operate in full symbiosis, strengthened by the application of a SQA process. Figure 2.4 completes the previous duality of reproducibility and reusability to include the SQA process as the third enabling requirement

to achieve reproducible research.

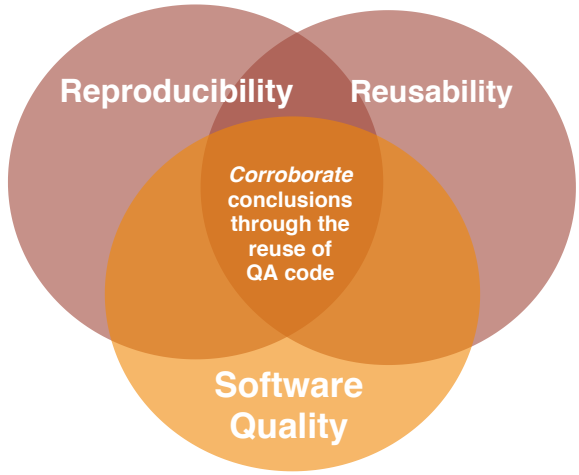

Figure 2.4: Software Quality Assurance as the enabling environment for achieving reproducible research

2.3 Conclusion

Similarly to data, software is a fundamental research object for achieving reproducibility in science, and accordingly, for the Open Science realisation. However, such perception has been undermined or not correctly conducted within the ongoing Open Science implementations, in particular the one being conducted by Europe. Accordingly, no formal regulation or policy about software has been considered and/or promoted, even though research software development evidences a series of inherent problems. Lack of awareness of software fundamentals, educational and endemic problems, as well as the suboptimal management of expert personnel are the most tangible types of problems existing in scientific environments.

Reproducibility in software is different from replicability, the most basic level

of scientific integrity, and reusability, which builds on an already existing code. The definition of reproducibility implies the use of an alternate and equivalent procedure to reach the same results, and thus, corroborate the original conclusion. Last section of this chapter has unveiled the need of exercising quality practices in software, following a SQA process, as the path to empower reproducible research through software reusability. The next chapter will elaborate on the foundations that support such a culture of software quality through the identification of the quality characteristics and the methodologies that lead to the successful implementation of such SQA process.

"Quality is not something you believe in, Quality is something you experience"

Robert M. Pirsig

3

Building a Culture of Software Quality

The outcome from the previous chapter suggests the establishment of a quality culture in research software development in the path towards the Open Science implementation. Even though several open educational initiatives are available to computing scientists, there is still a prevailing hesitation to adhere to Open Science standards that is deeply rooted in the scientific environments.

This fact has been confirmed in the preliminary steps of the Open Science implementation in the European research infrastructures. Several reports, commissioned by the European Union (EU), stress the need of a cultural shift in the research communities' mindset in order to embrace the practices enacted by the Open Science paradigm.

In order to lay out a culture that empowers the presence of quality processes

within the routine practice of research software development, and progressively build trust in such processes, it is precise to clearly showcase the beneficial aspects and identify the Software Engineering (SE) methodologies that enable them.

In this chapter, a discussion is carried out in the context of the main software quality practices that can impact on the scientists' experience, followed by an overview of the SE methodologies and approaches that will be the vehicles for implementing such practices. These methodologies play a decisive role in the use cases presented in the second part of the thesis, in particular in Act II "Where Theory Meets Praxis: the Implementation Process".

3.1 Why quality on research software?

The source code being built these days might be hard to be ran in future computing platforms, just as it might be the recreation of the old's code execution environment. According to the insights from the preceding chapter, steering the development of software towards *reusability* seems to be a more convenient and proactive way to make today's code more likely to be reproduced in the future. Reusability, as it was also stated, is enabled through the compliance with software quality practices that are being carried out throughout the Software Development Life Cycle (SDLC).

But the quality of software goes beyond reusability, as it aims at improving the software at different levels such as its *reliability* or long-term *sustainability*. Source code that has been developed following a minimum viable quality-based process is readily subject to modification. The chances of it being sustainable over the long run increase since it will be considerably quicker and cost-effective to modify or add new features to an existing software, rather than engineering new code from scratch. Sustainability in experimental sciences is constantly challenged by the evolution of theories, so software integrity shall be preserved when a new theory evolves or needs to be re-adjusted, according to the *Theory-software Translation* process [43]. Reliability ensures the consistency of the

scientific work by facilitating its replicability and promotes the software adaptability and portability to diverse environments.

All these quality-related topics, that are facilitated by the Software Quality Assurance (SQA) process, cooperate to mutual advantage. Quoting a classical reference in SE [44] "reusing well-designed and well-developed software will increase reliability and maintainability not only because the software has been previously tested, but also because it has been used successfully". Along the same lines, Bourque and Fairley, in their SE masterpiece *Guide to the Software Engineering Body of Knowledge* [41], corroborate the previous statement that reuse-based development "can enable significant software productivity, quality, and cost improvements".

Definition of Quality: the Metaphysics of Quality

Although conceptually substantiated in the SE literature, quality, as it is interpreted throughout this thesis, does not differ from its domestic and universal understanding. In modern times, quality is actively present as a means to set and ensure that the minimal standards are met for any product or service.

However, quality presents a particularly elusive definition subject to one's experience. From a metaphysical perspective, insights gleaned from Pirsig's *Zen and the Art of Motorcycle Maintenance* philosophical novel suggest that quality "exists always as a perceptual experience before it is ever thought of descriptively or academically". Hence, quality is a concept hard to be defined, but tightly aligned with the concept of *care*: "Care and Quality are internal and external aspects of the same thing. A person who sees Quality and feels it as he works is a person who cares. A person who cares about what he sees and does is a person who's bound to have some characteristic of quality". Consequently, quality demands intense focus, a "sincere desire to know what's best so we may create value".

The purpose of this work is not at all far from this conception, however in this case a systematic examination of quality for software will be given and demonstrated through empirical evidence.

Scope of quality in this study

Quality of software embraces all the concepts defined so far. From the three R's –replicability and reproducibility and reusability– of scientific computing to SE-related topics such as reliability and sustainability. Accordingly, the definition and implementation of a SQA process that improves the quality of research software is at the core of the present thesis, and thus, the next chapters will be progressively addressing the definition of the suitable quality practices and their further implementation within real research environments.

Throughout these experiences, beyond the immediate overall positive results obtained in the respective research projects, the participant computer engineers and scientists have become aware, through practice, of the importance of considering quality methodologies early in the SDLC. The SQA practices discussed in detail throughout this work, are indeed in line, and complement, the Open Science principles in such a way that skilled practitioners will definitely lead, by true conviction, the cultural shift required for the Open Science realisation.

Consequently, this work is based on the assumption that a better computing science rests on the shoulders of higher levels of compliance in the quality of software being produced.

Education on quality software for research

Several educational initiatives are tackling the academic-inherent problems in research SE from diverse perspectives. They are accessible as training programmes, scientific publications and manifestos or national facilities. As it would be impractical to undertake a comprehensive review of all the existing literature and initiatives, Appendix A.1 discusses the most representative contributions existing in global academic community.

In this regard, as one of the most prominent outcomes of the present work, Chapter 5 "Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software" will thoroughly present a baseline containing a comprehensive set of requirements and good practices to be considered throughout

Quality characteristic	Aim	Reusable	Reliable	Sustainable
Open and Accessible	Ability to access and reuse versions of the code	(✓)		(✓)
Collaborative and supportable	Project promotes community engagement and provides active support			(✓)
Readable	Source code is understandable	✓		✓
Testable	Seeks software reliability	✓	✓	✓
Secure	Implements security practices	✓	✓	✓
Discoverable	Software is uniquely and clearly identified	(✓)		✓
Portable and Interoperable	Software is stable on multiple platforms and relies on open standards	✓	✓	✓
Usable	Software provides documentation that makes the software buildable, installable and runnable	✓		✓

Table 3.1: The defining factors of quality in research software, where missing check marks mean *not required* and brackets indicate *optional* or *desirable* compliance. Several conclusions can be drawn from the table contents: i) both reusable and reliable are prerequisites for sustainability, and ii) sustainability requirements are considerably similar to those of quality software.

the SDLC.

3.1.1 The defining factors of quality in software

A SQA culture builds on the identification of the desired quality characteristics that shall be enforced to the software. To this end, a good approach is to leverage the work in the available literature [45, 46] and quality standards [47]. The strategy to compose the following enumeration of quality characteristics is inspired by those references, but eventually adapted in accordance to the intended scope of the present work.

Open and accessible

Doing research in an open manner is crucial for the timely questioning and discussion of the scientific breakthroughs, and the seamless dissemination of the scientific knowledge. Bringing your source code out in the open is a catalyst to accomplish the Open Science goal, and a pivotal requirement to address the currently existing reproducibility crisis. Free/Libre and Open Source Software (FLOSS) is then the primary prerequisite that enables the SQA culture.

In spite of the proven benefits, *going open* is still not a widely adopted practice in the scientific community, even in the case of publicly funded research projects. This trend must be reversed not only to sustain the continuous scientific progress or even to contribute to a fast-paced innovation, but also to strengthen the consistency of the research value, towards the path of full reproducibility in Science.

While the outreach capacity of software does not exclusively depend on its accessibility –e.g. it may be well more determined by the usefulness and innovative skills within the field of application–, the potential impact on the scientific field of application is far more likely to happen when the source code is exposed to the outside world.

Unrestricted access to the different objects that drive the software development practice has direct benefits on other quality aspects reviewed in this section. Source code accessibility implies the developer's commitment to clarity and transparency of the SQA practices throughout the SDLC process, which has the beneficial effect of building trust around the software product. This fact has the capability of building a community of collaborators for a fast-paced problem solving and provisioning of new features in the software.

Collaborative and supportable

Open Science advocates for data and code sharing. In some scientific computational environments there is the risk of insufficient funding for the maintenance and support costs that arise from a successful software development project.

Collaboration comes to the rescue as a way to share this effort among the workforce of volunteers that participate in it. Consequently, the success and popularity of a FLOSS project greatly relies on its capacity of engagement, and this latter characteristic definitely contributes to the long-term sustainability of the software.

Box 1.1: Incentives for voluntary contribution to FLOSS projects

Intrinsic incentives

- Human altruism or philanthropic nature
- Desire to belong to a team or community
- Craftsmanship model: satisfaction of getting something valuable, creative work admired by others
- Ethical values: alternative to propietary software
- Satisfy personal use-value of a product, scratching a "personal itch" in terms of software functionality

Extrinsic incentives

- Reputation: recognition among peers
- Rewarding: citation, access to better salary and/or employment
- Learning, education

Sources from [48, 49, 50]

There are well-known successful stories about the potential capacity of big collaborations in FLOSS projects. The oft-cited case of the Linux kernel is the most representative illustration of a strong community that is progressively being built upon the appeal of an open collaboration that develops a sustainable and highly competitive software solution [51]. In the research area, the benefits of open collaboration have been also empirically proven [48], asserting its

capacity to accelerate the discovery of research value.

Nevertheless, successful collaboration is obviously not granted by default for any FLOSS project. Several studies [52, 53] have measured the grade of inactivity of FLOSS projects. Abandonment or inactivity are usually originated by the developer's lack of time or interest, and undertaken by an inadequate handoff to interested developers, disregarding the advice of the already mentioned founder of the open source movement Raymond: "When you lose interest in a program, your last duty to it is to hand it off to a competent successor".

Collaboration poses another form of protection for the user of the software, as even in cases when the software reaches the end of support or goes into an inactive state, the user still has a way to extend its end of life. This is particularly helpful in academic environments where limited funding for setting up software solutions, via short-lived projects or grants, is commonplace. The fact that the research project ceases does not entail the discontinuation of the software product [48].

Without collaboration, software sustainability becomes a tougher task. Box 1.1 summarizes some of the intrinsic and extrinsic incentives that boost collaboration in software projects (note that competing for innovative leadership is not part of the incentives)

Readable

The feeling that source code is not readable is a deep-rooted belief of some scientists. In the early days of Open Science conception, Easterbrook stated in the Nature's article *Open code for open science?* [38] that releasing the code along with the scientific publication was not really needed since ***all but the simplest codes are impenetrable to non-experts***. This statement stems from the practical evidence of many computational scientists who are used to deal with unstructured code, and builds the conviction that in-depth programming skills are required for even trying to comprehend what the code is actually doing.

Often seen as an utopia, the code should be ideally *self-documented* [54], in particular meaning that the identifiers used throughout the code –variable,

class, method names– are self-explanatory [1]. Even if not reaching that stage, the code should at least be understood by the relevant scientific communities, otherwise the source code loses part of its value.

Readable code will not only facilitate the understanding of the method followed to obtain your scientific insights, but in the most prominent cases, readability can be a booster for external collaboration. Hence, as it was discussed before, it can lead to build a community –of users and contributors– around the software project.

Commitment to a SQA process is key to write understandable code. As it was also mentioned, new code contributions are not only done to active software projects, but as the maintenance of a FLOSS project usually entails a voluntary effort, it is susceptible to be discontinued at a certain point, regardless of its popularity. At this point, a "competent successor" [2] usually takes over the responsibility of the code maintenance. A smooth hand-off is only achieved if the code is readable, and thus, avoiding technical debt-related issues. How this debt will evolve is notably on the hands of the early developers.

Testable

It is rather a common practice in science to deal with software that shows unexpected behaviours, which leads to real struggle when trying to replicate results, or even formulate conclusions in the first place. Those unusable programs are usually composed of poor source code quality that lacks of a representative aggregate of test cases that would ensure the operation of the expected functionalities.

The ultimate goal of software testing is to increase the reliability of the systems being delivered to the users. However, testing cannot guarantee that a system performs accurately under all conditions, but only that it is not performing properly under *specific* conditions. As such, testing can only pretend

[1] Firm advocates of self-documenting code propose leaving the documentation in the background as a second-class object. In particular, one of the principles of the Agile manifesto [55] states that "Working Software over comprehensive documentation".

[2] According to Raymond, see [14].

that the uncovered errors are no longer present [3].

Furthermore, the economics of testing shall be carefully considered as well for every software project. On the one hand, inadequate investment may imply solving defects at later stages. Quoting from Perry [57], "it is at least 10 times as costly to correct an error after coding as before, and 100 times as costly to correct a production error". On the other hand, a generous effort may lead to increased project costs [58], not estimated in the project design, and delays in the release dates [59]. Therefore, measuring the cost-effectiveness of the testing process does not only imply stopping at the optimum point where the cost of testing does not exceed the value obtained from the defects uncovered, but rather focusing on the valuable features first [60].

Secure

Granting the secure operation of the software is a fundamental objective of testing. Independently of the size and popularity of the project, developers must consider security flaws with the highest priority. This entails immediate action to publicly-disclosed vulnerabilities and a preventive assessment, tackled through the SDLC, to uncover and fix common security bad practices to prevent software to be compromised by attackers. Security testing is no different from common testing and occasionally security flaws cannot be spotted during the *closed-doors* environment that characterizes the SDLC. As a result, policies and procedures shall be in place in order to react promptly to security issues being reported by external users. Nevertheless, a great deal of software maintainers do not usually have enough training to perform an accurate software security assessment. It is a complex domain that requires from a high level of expertise. Again, the larger the open collaboration, the more likely to reckon on security experts. Conversely, small projects might face this problem, but as it will be shown in the second part of the thesis, there are automated security testers that

[3] C. Kaner el al. [56] identify three main reasons why complete testing is impossible: 1) Domain is too large to test, 2) Too many possible paths through the programs to test, and 3) User interface issues are too complex to be completely tested.

require no prior experience to be leveraged.

Discoverable

Despite the traditional thinking of software being a second class research object still persists, recent scientific literature unanimously consider software as a value object, equally as important as the resulting scientific journal article or the data sets used and produced throughout the scientific process [61]. In fact, journals are increasingly enforcing the unique identification of the research publications where the software has played a key role in conducting the scientific insight. The achievement of higher levels of reproducibility in science is behind the current scholarly trend of revising the software as a first class product of research.

Likewise the data, the existing path to ***achieve the full identification of the software is through the use of metadata***. Hence, research software improves its discoverability, which potentially leads to software reuse. Software citation –if any– is commonly tackled by referring to the journal article or scientific paper that describes it, rather than citing the software itself. This leads to inconsistencies since the version used might not match with the one being described in the cited paper. ***Persistent and unique identifiers make software citation viable in journal publications, especially if such identifier is mapped to a specific release or version of the software***, thus facilitating the task of replicating the scientific experiment.

Portable and interoperable

Portability of software is the capability of the software to be usable on multiple platforms or environments. As it can be inferred from this definition, no modification in the code is *allowed* for a software system to be ported to a different platform. Consequently, portability must be granted during the SDLC and it is enabled through the use of testing environments. Software can be then validated for each one of those environments and adapted accordingly. Portability is an operational requirement for reproducibility.

Software interoperability connects two different systems so that they work together. Interoperability in software is achieved by the use of open standards, which make the software usable and compatible with coupled technologies. As such, open standards create a fair market that prevents from *vendor lock-in* situations. Unlike the FLOSS projects, private companies developing their own software solutions –remember the *Cathedral* model from [14]–, may decide not to follow the specification of an open standard. Instead, source code openness provides the flexibility to allows any contributor to enhance the code, and thus, provide support to any given standard [62].

Usable

The ultimate goal of software is to be used by the interested stakeholders. In the scientific domain, usability requirements are often mistakenly circumvented based on the assumption that the software is exclusively used by the group tackling its development. No matter the target audience, the software needs to be provided with guidance for building, installing, running and harnessing. Complete documentation is key for the software adoption.

3.2 The path to software quality engineering

This section assesses software quality from a technical perspective through the observation of SE processes and methodologies. The bottom line is to explicitly situate the objectives of a SQA process as the groundwork for the subsequent chapters in the second part of the thesis.

Quality management in software engineering

According to Bourque and Fairley's *Guide to the Software Engineering Body of Knowledge* [41], the de-facto SE reference, the practice of SE is comprised by a set of knowledge areas, among which the so-called "Software Quality" is the most relevant for the interests of the present thesis. According to this work,

software quality is a transversal area, tightly coupled with related areas such as "Software Maintenance" and "Software Testing".

Conceptually, the management processes of software quality boil down to the i) ***definition of the quality requirements for the specific software product to be built***, the SQA process, followed by the ii) ***implementation and operation of the means –i.e. procedures, technologies and tools– to achieve those***, managed by the Verification and Validation (V&V) processes.

3.2.1 Software Quality Assurance

According to the SE practices, the SQA process is driven by a plan, or Software Quality Assurance Plan (SQAP), usually part of deliverable documentation in research projects, that ensures the quality targets are precisely defined. A SQAP compiles the quality activities and tasks, schedules, metrics and risk assessment, estimation of resources, technologies and tools that enable the SQA process. The ultimate goal is to seek the achievement of the software quality objectives and the stakeholder satisfaction, so these processes build upon the end user requirements.

The quality characteristics previously defined in Section 3.1.1 "The defining factors of quality in software" provide the guidance that helps identifying the concrete requirements to be compiled in the SQAP and enforced during the SQA process. This task will be carried out later on in Chapter 5 "Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software".

3.2.2 Verification and Validation processes

The V&V processes contribute to build quality into the system during the SDLC. While planning V&V is part of the software quality management processes, it also drives the practical implementation of the SQA process, so it is often considered within the software development and testing area.

V&V are commonplace concepts in SE literature, but these terms are often –and mistakenly– used interchangeably in practice [63]. Indeed, both processes

serve different purposes since *verification* is linked to the early stages of the SDLC, ***focusing on building the software correctly***, while *validation* is commonly placed at the end of the development process, providing ***"evidence that the software and its associated products satisfy system requirements allocated to software at the end of each life cycle, solve the right problem, and satisfy intended use and user needs"*** [64].

Box 2.1: V&V processes in research reproducibility

As it was discussed in Chapter 2 "The Role of Software in Open Science", V&V processes are commonplace in the context of scientific computing reproducibility. Here, the verification addresses the replication of an experiment, while validation refers to the evaluation of the experiment's result to prove that the author's conclusions are justified [65].

Static and Dynamic analysis

The practical application of the V&V processes consists in the implementation of the static and dynamic analysis of the software. Whilst the static analysis techniques evaluate the software by examining the code without actually executing it, the dynamic analysis evaluates the software at runtime.

Static and dynamic analysis give rise to additional dualities in SE literature, such as the popular concepts of white and black box testing, which are complementary types of testing that focus, respectively, on the code structure and the functionalities provided by the software. Figure 3.1 summarizes all the aforementioned concepts existing within the V&V processes.

It is important to remark that it is not in the scope on the current work to build on the knowledge base of software testing, such purpose is thoroughly covered by the existing literature on SE. Appendix A.2 covers the types of testing that are relevant for the appropriate understanding of the prospective identification of SQA requirements elicited in Chapter 5 "Wrapping-up: The definition

of a Software Quality Assurance baseline for Research Software". Accordingly, it is highly recommended to be familiar with those definitions before proceeding to the Act II "Where Theory Meets Praxis: the Implementation Process". Additionally, Figure 3.2 situates each type of testing in accordance with the V&V processes.

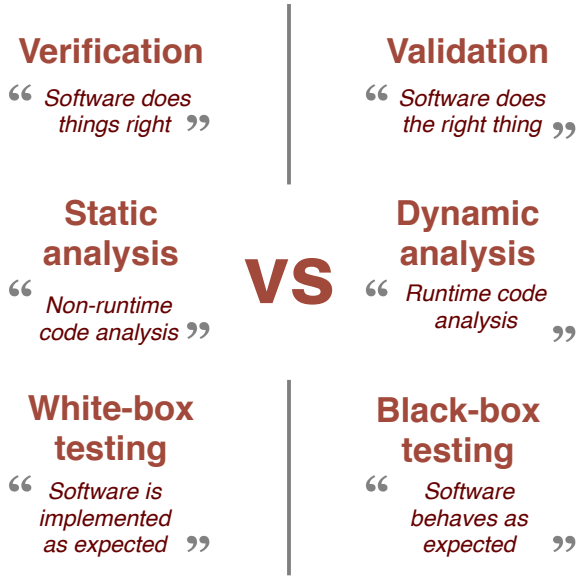

Figure 3.1: The duality (and synergies) in V&V processes

3.3 The DevOps culture: automation to enable quality

Automation brings faster completion of the SQA process when a new change is added to the source code. Furthermore, this optimization of the time-to-deliver new changes enables the rise of new requirements that can be progressively added to the SQA process. However, relying overly on automation may intro-

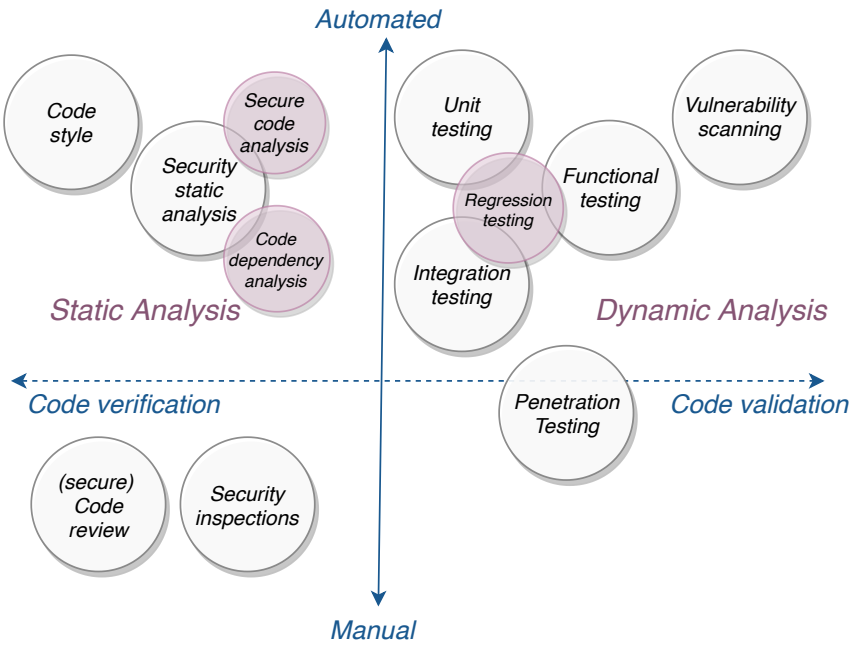

Figure 3.2: Categorization of the different types of testing that are relevant within this study. They are classified according to their suitability of automation and grouped within the V&V processes.

duce risks in the software development chain, so manual or human observation offers an undeniable value along the SQA process.

The SE methodologies foster a culture of continuous application of V&V processes by harnessing automation are most relevant for the challenges ahead within the present work. Having automation as the game-changer, increased levels of quality in software development, sustainability and delivery can be reached.

Testing automation

Test automation is gaining momentum as a way to decrease the costs associated to software testing tasks. *Efficiency* gets improved as automation optimizes the execution time of testing, maximizing the test coverage as more tests can be performed in less time [66]. The augmentation of the test coverage strengthens the reliability of the end product, reducing the number of defects present.

Automation also increases the overall *effectiveness*, avoiding the risk of human errors and achieving repeatability. This is particularly useful to reduce the regression risk by finding defects in the modified, but previously working, functionalities of the system [67].

However, test automation does not always supersede manual testing. According to a number of studies [68, 69, 70], not all the testing tasks can be easily automated, such as those requiring extensive knowledge in a specific domain, or they require a costly maintenance. In such cases, manual testing complements automation, but generally speaking, automation shall be the end purpose in testing.

Introducing DevOps

DevOps is the reference paradigm, in recent times, when it comes to build agility and efficiency throughout the SDLC. As the name suggests, DevOps theorizes and provides practical solutions that aim at unifying traditionally confronted parties, i.e. software development (Dev) and infrastructure operation (Ops) teams, in order to tear down the so-called "wall of confusion" [71] built over the years between them.

To this end, DevOps puts emphasis on the establishment of a SQA culture to drive the development phase that eventually builds trust in the operational side. As a consequence, DevOps culture is commonly sketched as illustrated in Figure 3.3. It is indeed the SQA process that enables the DevOps culture, but automation is the vehicle to achieve it through the fulfillment of a high amount of quality characteristics. Automation optimizes the execution of the SQA process

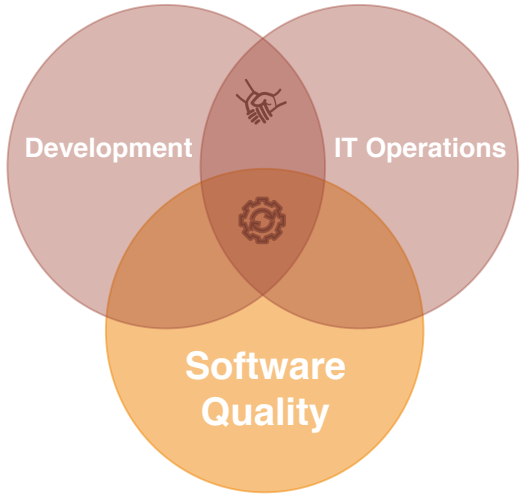

Figure 3.3: In the DevOps culture, an effective collaboration and communication between the development and operations teams is only accomplished by assuring quality in the software produced. Automation enables the realisation of the SQA process that builds the required level of trust on both teams.

and makes it viable to be continuously applied in a change-driven scenario. The transparency of the SQA process allows further confidence towards production infrastructure operators.

Commitment to agility

DevOps builds on the Agile software development approach that emerged as a response to previous models characterised by their inflexibility, such as the Waterfall methodology [72]. There is indeed an historical connection between DevOps and Agile that traces back to the 2008 Agile Conference, where the DevOps term was first coined.

Agile –see Appendix A.3– and DevOps are cultural movements that complement each other. Both promote the incremental delivery of quality software, but, whilst Agile offers guidance with regards to the managerial skills of the

software team itself, DevOps goes beyond extending this collaborative dimension to the IT operations teams, leveraging the spread of automated deployment and cloud-based provisioning technologies.

Challenges in adopting Agile are at the cultural and educational level, but DevOps adds to those the complexity of its varied forms of practical implementation. As it will be demonstrated throughout the chapters in the second part of this thesis, there is no *one-size-fits-all* when it comes to implement a DevOps approach. The generic approaches to DevOps are presented below.

Continuous integration and delivery

The practice of DevOps can be applied at different stages in the SDLC, usually encompassing the integration, delivery and deployment. Due to its nature, not all the existing software facilitates the execution of the entire cycle, however the flexibility of the different DevOps approaches allows tailored strategies to be put in place.

Continuous Integration (CI) is primarily connected to the static analysis of the source code. This is not to say that dynamic testing cannot take part at this stage, which implies the involvement of the more advanced and subsequent approaches in the DevOps chain –i.e. delivery and deployment–, but rather that the static analysis is a must at this point. The idea behind CI is to integrate developers' work as early, often and as safely as possible, relying on automation tools, which will be discussed in the next section.

Continuous Delivery (CD) is the next stage in the DevOps realisation. Although there is no general consensus in the literature, the prevailing view is to consider the automated deployment as part of the CD process. This is indeed the cornerstone of the agility of the DevOps implementation, as it implies the automated resource provisioning and deployment. According to DevOps principles, both tasks shall be automated leveraging Infrastructure as Code (IaC) or Continuous Configuration Automation (CCA) solutions, which use machine-readable definitions to express the full software provisioning process.

Hence, once the source code is statically tested and subsequently deployed

in an isolated testing environment, additional dynamic analysis testing is performed over the system on execution. If successful, the software is packaged and delivered. Note that, as it was mentioned before, the possible approaches of DevOps are diverse, so that there is room for more complex scenarios, such as the automated deployment in production environments. In any case, the ultimate goal of a DevOps implementation is to have an automated, end-to-end *pipeline* tackling the CI and/or CD phases, aiming at reducing the development-to-production time without negatively impacting on quality.

The CI infrastructure

DevOps has computing industry as its primary influence, and as a result it is often associated with benefits on metrics such as Time to Market (TTM) and Return on Investment (ROI), measuring respectively the duration of the delivery cycles and the investment's gains relative to its cost. The popularisation and widespread industry adoption [4] of the DevOps culture motivated the emergence of tools that facilitated the implementation of the aforementioned approaches.

In terms of availability, the myriad of DevOps tools can be either categorized as cloud-based or on-premise. The former ones are particularly suitable for individual projects with standard requirements with regards to the Continuous Integration and Delivery (CI/CD) capabilities. As the requirements grow, tools that facilitate on-premise deployment are increasingly needed. Jenkins CI [74] is the most representative CI tool to meet the specific needs of all-size projects, as it will be the reference technology for the practical insights exposed hereinafter.

3.4 Conclusion

Quality of software embraces all the concepts defined so far, not only as a result of the etymology of the word *quality*, but from the more technical perspec-

[4] According to the 2019 State Of Devops Report [73], high performance companies like Google, Amazon and Netflix use DevOps practices to deploy software thousands of times per day.

tive given by SE literature. Hence, quality covers the realisation of reusability, identified as key aspect for the accurate reproducibility of computational experiments in Open Science, as well as the additional goals commonly pursued when dealing with software, such as reliability and sustainability.

A set of characteristics have been identified in order to better understand what software quality actually encompasses. These characteristics shall be unambiguously mapped to requirements in order to enable the systematic implementation of quality software. This task is managed by the SQA process that, as it will be shown in Chapter 5 "Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software", procures a policy plan to drive the SDLC.

The V&V processes are particularly suitable to drive the SQA process, covering both the code analysis, or static, and assessment of the runtime execution, or dynamic analysis. These process are tightly coupled with the final reliability of the software, so it is crucial to be diligent at this stage, but also attentive not to overestimate the capabilities of testing, as 100% reliable software is not doable.

Automation definitely helps in driving the SQA process, otherwise difficult to tackle according to its complexity or the volume of quality requirements. Besides, the costs associated to software production are mitigated. In SE-related literature, automation is usually associated to testing purposes, but the aim in this study is to widen the scope to all the SDLC activities that seek quality in the software. DevOps culture and related implementations, such as CI/CD, will be the landmark in the SQA approach being discussed in the next chapters.

4

Software Quality to drive the delivery of services in the European Open Science Cloud

In the European Union (EU) context, research *e-Infrastructures* [1] provide the computational capacity for multi-disciplinary Research Infrastructures (RIs) that meet their *e-needs*. The Open Science implementation draws on their of-

[1] According to IGI Global academic publisher, an e-Infrastructure is defined as "short term for Electronic Infrastructure, i.e., all Information and Communications Technology (ICT) based resources (distributed networks, computers, storage devices, software etc.) and support operations which facilitate the collaboration among research communities by sharing resources, analysis tools and data."

ferings and availability in order to deliver the resources and services that allow scientists to perform their research, and thus, e-Infrastructures appear as the driving force for the establishment and success of the Open Science movement.

The usability of the e-Infrastructures is heavily dependent on the quality of the services offered to the researchers, especially in terms of stability and reliability. Here, the quality of the underlying software that makes up a service is crucial for achieving these goals. The software quality concepts from the previous chapter can be applied to any software product, regardless of the scope of application. Whether it is e-Infrastructure enabling software –i.e. the software that provides the functionality of thematic services within the RIs–, or the standalone software used to conduct a particular scientific experiment; exercising the quality of software shall be a common objective within every development and maintenance effort.

This chapter situates the Open Science implementation in the European research scene, which shall be underpinned by the extensive experience of the existing European e-Infrastructures in delivering services to the RIs, and therefore, for the scientific communities' exploitation. The guiding premise is the relevance of the software in the operational accuracy and functional suitability of such services, either being e-Infrastructure enabling or fit-for-purpose research services. This chapter provides a suitable contextualization for the incoming chapters.

4.1 Enabling Open Science in Europe: the European Open Science Cloud

The European Open Science Cloud (EOSC) is the response to the Open Science realisation within the European research ecosystem. The EOSC was officially announced by the European Commission (EC) with the adoption of the Digital Single Markets strategy on May 2015 [75]. The goal is to build a unique access point to an open, trusted and federated environment that supports the research

needs of 1.7 million European researchers and 70 million professionals in science and technology. The implementation of the EOSC shall be facilitated by the federation of existing generic, or *horizontal*, e-Infrastructures at the national level together with domain-specific, or *vertical*, RIs [76].

4.1.1 The Horizon 2020 Framework Programme

Horizon 2020 is the biggest EU Research and Innovation programme to date with €80 billion of funding available over 7 years, from 2014 to 2020. An overarching goal of the Horizon 2020 is to deliver an EOSC that "truly supports interdisciplinary research and Open Science". The EC used the Horizon 2020 Work Programmes, in particular through the INFRAEOSC dedicated call, to consolidate the implementation of the EOSC, planned to be realized by the end of the programme, in 2020. To this end, the Horizon 2020 particularly promoted the integration of existing e-Infrastructure platforms and RIs to support the development of cloud-based services for Open Science.

A series of EOSC-enabling projects were progressively funded in order to integrate the most relevant and leading technology into the future EOSC. The ongoing EOSC-hub [77] project, started in 2018, is driving the integration of the initial set of shared resources, and as a result of the outcomes of the first months of the project, it was the principal benefactor of the EOSC Portal launch back in late 2018. The EOSC-hub project is primarily working on delivering technologies and services from the major research e-Infrastructures, such as the European Grid Infrastructure (EGI) Federation and EUDAT Collaborative Data Infrastructure (EUDAT-CDI), and Technology Providers (TPs), such as INDIGO-DataCloud (INDIGO) project.

4.1.2 The foundational elements of the European Open Science Cloud: e-Infrastructures and Research Infrastructures

In the European Union context, a RI encompasses those resources and services required by the research communities in order to undertake the continuous innovation challenges. The e-Infrastructures are an integral part of them, covering the data and computing resources, as well as the communication networks.

In the European research context, the EINFRA call, under the Horizon 2020 framework, has enabled a constant expansion of the e-Infrastructures in order to offer specialized data-driven and computer-intensive capabilities for science and engineering. Hence, e-Infrastructures are playing a leading role in the European research context as almost all large-scale research activities include or are supported by several e-Infrastructure components [76]. The e-Infrastructures have become very experienced in delivering distributed computing services to researchers, and as such, shall be key players in the specification and the setup of the EOSC.

4.1.3 The roadmap to the implementation of an European Open Science Cloud

The EOSC is an evolving federation of European research data, services and infrastructures in compliance with the Open Science principles. The EOSC has been defined as a long-term process that will provide user-driven services to European researchers –covering both the academia and industry–, supported by an underlying infrastructure that provides the required advanced computing, networking and data management capabilities. Throughout this process, which is currently ongoing, the EOSC implementation has gone through two different stages, clearly separated by the official launch of the EOSC Portal, which took place in November 2018.

Stage I: EOSC vision and launch

Appendix B.1 presents the main checkpoints that took place during the EOSC vision stage. This stage covered the consultation phase, targeted to experts and relevant stakeholders in the European research ecosystem, and the definition and roadmap implementation of the preliminary phase of the EOSC. Two High Level Expert Group (HLEG) reports [78, 79] and the EOSC Declaration [80] were the main outcomes of the consultation, based on which the EC defined the EOSC Roadmap implementation [81] of the preliminary phase of the EOSC.

Box 1.1: Action lines in the EOSC Roadmap

1. Build a federated *architecture* that remedies existing RIs' fragmentation.
2. Promote a *data* stewardship culture through the FAIR principles.
3. Elaborate a catalogue of interdisciplinary and user-driven *services*.
4. Guarantee universal *access* through the EOSC Portal.
5. Definition of the *rules* that set out the rights, obligations and accountability of the different stakeholders, mainly service providers and users.
6. Design of a representative, multi-stakeholder *governance* model.

The EOSC Roadmap defined six main action lines to be pursued in the incoming years, determined the lines of funding through the Horizon 2020 Work Programme 2018-2020 and the timeline with the milestones to be reached. The six action lines are enumerated in Box 1.1 and built on the foundation of a federated environment of e-Infrastructures resources and research services with unique access through the EOSC Portal. The range of services shall be representative of the multiple scientific disciplines existing in the European research ecosystem, and accessible through the EOSC catalogue once compliant with the so-called Rules of Participation (RoP). In addition, data management services offered through the EOSC Portal shall promote and allow the management of

data according to the FAIR principles.

This first stage concluded with the first release of the EOSC Portal, the main entry point for the EOSC audience, where all the integration efforts, in terms of data and services, are progressively being added.

Phase II: from EOSC launch to sustainability

The next steps towards the sustainability of the EOSC build on the integration work continued through the ongoing EOSC-hub project, being complemented by specific funding from the INFRAEOSC call. The Strategic Implementation Plan [82] delivered in 2019 emphasized the liaisons between projects of the call. This document established the priority of providing a set of recommendations concerning the implementation of an operational, scalable and sustainable EOSC federation after 2020.

Later on in 2019, the EOSC Work Plan 2019-2020 [83] outlined the activities, timelines and key inputs to be tackled until the end of 2020, where a new governance structure will be set up.

4.2 Service maturity assessment within the European Open Science Cloud

The EOSC's most visible outcome is the progressive federation of multidisciplinary data infrastructure services that exist within the European research boundaries. Those services are to be readily accessed by the researchers through a marketplace, the EOSC Portal, and as such, quality-related aspects such as the usability, reliability or functional accuracy, shall be assessed beforehand. As discussed in the previous section, the EOSC-hub project is driving the preliminary phase of an operational EOSC. To this end, the project is progressively tackling the integration process of the most mature e-Infrastructure and RI services, both internal to the project –the thematic services, competence centers and business pilots planned for inclusion in the description of activities– and

also external services, coming from the European RI landscape, through the Early Adopters Programme (EA).

Technology Readiness Levels (TRLs)

TRLs are a type of measurement system used to estimate the maturity level of a particular technology. The outcome of the evaluation is a TRL rating, that ranges from TRL-1, the lowest maturity, to TRL-9, which corresponds to the highest maturity.

Since the Horizon 2020 Work Programme 2014-2015 [84], the EC officially adopted the NASA's TRLs in order to estimate the ***maturity of a technology***. Accordingly, based on the EC mandate, EOSC-hub adopted the TRL scale in order to define the minimal requirements for inclusion in the catalogue, setting the TRL-8 as the reference for production technologies. Box 2.1 provides the definition of TRL-8 and TRL-9.

Box 2.1: TRL-8 and TRL-9 definitions

TRL8: System complete and qualified End of system development. Fully integrated with operational hardware and software systems. Most user documentation, training documentation, and maintenance documentation completed. All functionality tested in simulated and operational scenarios. Verification and Validation (V&V) completed.

TRL9: Actual system proven in operational environment Fully integrated with operational hardware/software systems. Actual system has been thoroughly demonstrated and tested in its operational environment. All documentation completed. Successful operational experience. Sustaining engineering support in place

Potential shortcomings of a TRL-based service assessment

The TRLs are expressed as high level descriptions. A *technology* is conceived as a combination of hardware and software deployed in an operational environment,

which at first glance is quite in line with the *service* concept, as it is used in the consultation documents of the EOSC implementation. But from a Information Technology Service Management (ITSM) perspective [85, 86], a service is a "means to deliver value to a customer", and accordingly, ***a technology, as described by the TRLs, misses the user component***, especially the extent to which the user support shall be covered.

This fact was actually acknowledged within EOSC-hub deliverables, where the suitability of the TRL system for use within the context of operational service delivery was questioned [87] [2]. Consequently, the TRL-based service assessment was enhanced to include the evaluation of both operational and technical aspects. On the one hand, the adoption by the project of a lightweight ITSM framework, FitSM [88], helped to cover the operational requirements. However, not all the services from EOSC-hub required the same level of integration, being the operational services the ones with higher ITSM requirements and the research enabling services the ones with lower requirements. On the other hand, the technical requirements were elicited following an agile framework that identified the technical gaps that need to be addressed for a successful integration within EOSC-hub. The identification of the operational and technical requirements improved the TRL-8 and TRL-9 assessment.

Software quality as the key indicator for service assessment

But ***the remaining issue with the TRL assessment, not solved by EOSC-hub approach, boils down to the lack of accuracy in the requirements being formulated, and, more importantly, each level do not provide the means of verification to ensure how its requirements can be fulfilled***. Hence, having "most documentation completed" or "thorough demonstration and testing in an operational environment" leads to confusion and divergent interpretations. The immediate consequence is that the owners

[2] The EOSC-hub's D4.1 deliverable *Operational requirements for the services in the catalogue* states that "TRL has its limitations as it is usually used to describe the maturity of underlying technologies rather than the delivery of them in the form of a service to end users".

of the technologies can only but vaguely estimate the appropriate TRL level, which can be hardly proved through the EOSC service integration process. In this regard, the limitations of the TRL scale were also highlighted in documents produced within the European RIs when trying to ascertain the minimum levels of maturity for the software being deployed in such infrastructures [89].

This lack of accuracy does not contribute to building trust in the research users that eventually will rely on the EOSC services to tackle thier work. In the worst case scenario, users' expectations of stability cannot be met because the maturity of those services is overrated. If TRL-8 is considered production-level, the associated requirements must be unambiguously detailed and the quality of the service shall be underpinned by the transparency of the previous software development and piloting stages the service has gone through. Thus, the expected functionalities of the service are readily visible by the users and have been proven to work before reaching the production level.

Chapter 5 "Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software" presents such a clear definition of quality requirements that drove the Software Quality Assurance (SQA) process that managed the development of services for European e-Infrastructures, also discussed in subsequent chapters. The results obtained definitely contribute to increase the accuracy of service maturity measurement, and to develop fit-for-purpose services for the involved research communities.

4.3 Quality-aware e-Infrastructures as the guidance for the European Open Science Cloud implementation

The 2018 EOSC implementation roadmap [81] concluded that the EOSC architecture would build on the existing e-Infrastructures and RIs, as an "soft overlay to connect them and making them operate as one seamless European research data infrastructure". Accordingly, e-Infrastructures and EOSC benefit mutu-

ally since the e-Infrastructures already have the experience in providing cloud services to researchers, while the EOSC will consolidate their federation, thus removing the existing fragmentation in the European infrastructures, which is a reality actively highlighted during the EOSC consultation phase. Indeed, the aforementioned 2018 EOSC implementation roadmap document underlined the key role of the existing e-Infrastructures in Europe as the baseline for building the EOSC and cover the whole life-cycle of the services, from planning to delivery.

4.3.1 Software quality for e-Infrastructure operation and exploitation

As previously introduced, the European e-Infrastructures have traditionally confronted similar issues in terms of stability and functional suitability of the services exposed to their target scientific communities. From pre-Horizon 2020 programme's projects that envisioned the implementation of Grid technology-based infrastructures as the new paradigm of distributed computing, to the current days of Cloud computing models, the continuous need of new tools and services to match research requirements has been addressed by the EC through dedicated software development projects for e-Infrastructure creation, operation and management. The result of this investment was that the reliability and adequacy of the services provided by such e-Infrastructures was progressively enhanced following novel and state-of-the-art software engineering practices. Hence, building on the knowledge base built and the expertise gathered throughout these past and ongoing projects is an optimal approach for the service delivery in the EOSC.

In the years of the inception of the EU framework programme Horizon 2020, the main drivers of the culture of innovation, as far as the software engineering in research is concerned, already shifted their attention towards the future quality and sustainability of the software [28]. Such a change of mentality was notably motivated as a consequence of the feedback obtained from stakeholders,

such as the e-Infrastructure users, who were familiar with the past instability issues. This trend permeated in the Horizon 2020 programme's EINFRA call, as reflected in the Work Programme 2014-2015, by demanding "services to ensure the quality and reliability of the e-infrastructure, including certification mechanisms for repositories and certification services to test and benchmark capabilities in terms of resilience and service continuity of e-infrastructures".

4.3.2 The missing element in the European Open Science Cloud equation

But as discussed in previous section, the advent of the EOSC did not bring new incentives towards the production of quality software. Much of the focus shifted to data stewardship, leaving the software –previously devised as a crucial element both for the e-Infrastructure and RI services' operation and exploitation– as a second class citizen. This fact is reflected through the analysis of the presence of software in the EOSC design documents, as shown in Box 3.1, which complements the already-identified undervaluation of software as found in the EC's Open Science Policy Platform (OSPP) advice document [26], highlighted in Chapter 2 "The Role of Software in Open Science".

Therefore, there is no guidance, incentive or rewarding structure defined within the EOSC implementation roadmap that drives the appropriate development and maintenance of the software that enables the accurate operation of the services that will be integrated and offered through the EOSC on a prospective basis. Software and services require active maintenance [3] and the EOSC must define benchmarks for excellence in software, not only to protect it as an essential asset for knowledge management and sharing [90], but as a means of protection for the EOSC audience. Otherwise, the EOSC will confront similar shortcomings in service exploitation by researchers as the ones faced by the

[3]The 1st HLEG report [78] points to a cultural clash among e-Infrastructure providers and scientific domain specialists that leads to a shortage of data expertise in Europe. The document remarks that "scientists expect that Open Source, project-funded software tools will stay magically updated, online; they will continue to produce and consume new data types, even when the project generating them is long-gone"

e-Infrastructures in the past.

Box 3.1: Role of software in the EOSC implementation

EC Communication in 2016 defines the EOSC as the access point of a catalogue of interoperable services to manage and share data. Among the six main lines of consideration for the EOSC implementation, ***services are expected to be "open and seamless"***, but no mention to the quality of the services whatsoever. Conversely, data is required to be FAIR.

First HLEG report in 2016 bolsters the adoption of the RoP as the criteria for service adoption in the EOSC. ***Software is timidly mentioned as one of the "data related elements" that enable reproducibility in Open Science***.

EOSC Declaration reports that "software sustainability should be treated on an equal footing as data stewardship", but still only mentioned in one out of the 33 requirements for the EOSC implementation.

EOSC implementation roadmap does not implicitly make any reference to software. The action line about the "development of the initial catalogue of services to be provided via the EOSC" fosters the integration of existing services offered by the e-Infrastructure providers –such as EGI, EUDAT-CDI and INDIGO–, with the RoP as the only criteria for service onboarding in the EOSC catalogue.

Second HLEG report 2018 makes specific reference to software developers, and service providers, within the relevant actors for the preliminary implementation of the EOSC, coined as Minimum Viable Ecosystem (MVE). Moreover, the report states the need of an ***EOSC-Ready*** certification for software in order to promote the long-term sustainability of the EOSC operation and give credit to innovative software developments. This is the first time that the role of the software in the service provision of the EOSC is accentuated, promoting the development of ***"open, sustainable, versioned, documented and energy consumption aware software"***.

Strategic Implementation Plan includes software as one of the driving prin-

ciples for the implementation of the EOSC. However, whilst software is targeted to "play a specific role as enabler of services and interoperability", the data are ascribed as the only first-class citizen. According to the plan, software requirements are narrowed down to two: open source and open for contributions.

4.3.3 Featured e-Infrastructure enabling initiatives in Horizon 2020 Programme

The chapters that follow, clustered around the Act II "Where Theory meets Praxis: the Implementation process", illustrate the impact that the implementation of a SQA process has on the delivery of services both for e-Infrastructure operation and for scientific experimentation, which might eventually contribute to the EOSC. In order to contextualize the insights described in the incoming chapters, the following lines introduce the aforementioned e-Infrastructure development projects.

EGI Federation

The EGI Federation federates computing and data resources, mainly hosted in Europe, into a publicly-funded e-Infrastructure that offers a catalog of advanced computing services, based on Cloud and Grid technologies, and support for research and innovation. The audience of EGI ranges from multidisciplinary scientific communities and research infrastructures to industry. At the time of writing, the EGI federated e-Infrastructure provide access to more than 1M computing cores and 900 PB of storage [91] via High-Throughput Computing (HTC) offerings –enabled through the Grid technology– and 21 cloud compute providers through the EGI Federated Cloud.

Back to the EGI's inception in 2010, the EGI acronym stood for *European Grid Initiative*, as at that time the Grid was the emerging distributed computing paradigm. As a result of the governance model shift from the preceding Enabling Grids for E-sciencE (EGEE) project [92], EGI was consolidated as the new

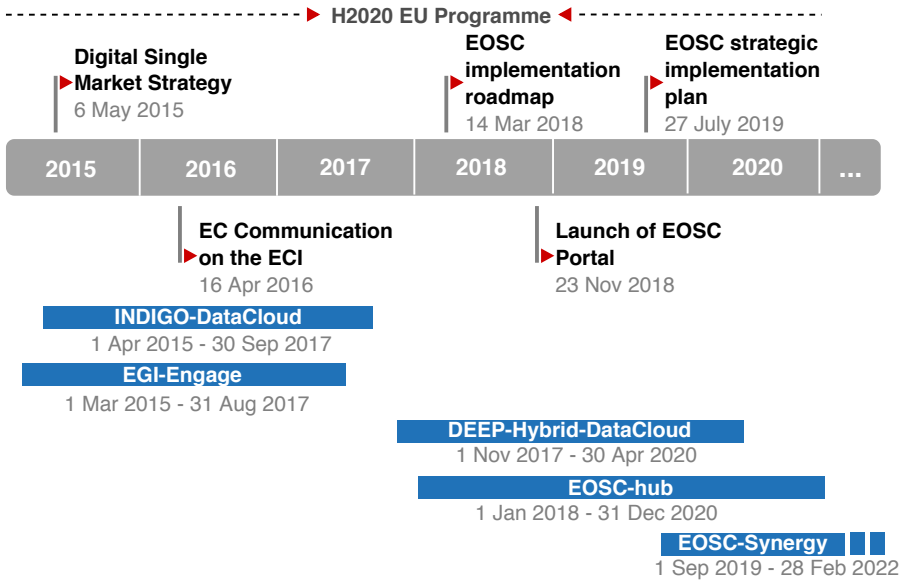

Figure 4.1: Lifespan of the most representative Horizon 2020 e-Infrastructure enabling projects within the context of the current work. The key milestones in the EOSC implementation are highlighted in the figure.

coordinating body of a pan-European Grid-based e-Infrastructure build upon National Grid Initiatives (NGIs), and interoperable with other Grids worldwide. EGI was subsequently *scaled* from initiative to infrastructure in order to reflect the transition towards a more sustainable effort [93]. This fact set the path for the establishment of the EGI Foundation as the coordination body of the EGI e-infrastructure. Until 2018, the EGI activities were carried on through the lifespan of two EINFRA projects, EGI-InSPIRE and EGI-Engage, which from that point onwards were continued under the scope of EOSC-hub.

The EGI Federation infrastructure is enabled by the services and software developed by diverse technology providers, and delivered through two software distributions, Unified Middleware Distribution (UMD) and Cloud Middleware Distribution (CMD), dealing with the aforementioned HTC and Cloud offerings,

respectively. The UMD and CMD products are validated before being deployed in the infrastructure by means of the EGI Software Provisioning Process (EGI SWPP), which does not interfere in their software development process, but focus instead on the quality assurance of the software artefacts contributed by those technology providers. To this end, the services are deployed and tested, according to the requirements in the EGI Quality Criteria (EGI QC) [94], as a preventive measure against prospective malfunctions once in the production infrastructure. There are currently around 60 products [95, 96] being maintained through UMD and CMD distributions. Chapter 8 will dive into the details of the EGI SWPP modernisation as part of the outcomes of this thesis.

INDIGO-DataCloud (INDIGO)

The INDIGO [97] consortium included 26 partners from 11 different countries. The project duration was limited to a 30-month period, starting in 2015 and concluding in late 2017. The primary goal of the project was to develop advanced cloud-based tools, applications meant to be used within the European e-infrastructures, such as the EGI Federation. Accordingly, it was a major effort to incorporate the cloud technology capabilities within those e-infrastructures in order to bring them within reach of the researchers. To this end, the solutions developed built on well-known open source cloud frameworks and included enhancements at the different infrastructure –covering computing, storage and network technology gaps–, platform and software as a service (IaaS, PaaS & SaaS) cloud levels.

The project delivered two main software releases, each comprising about 40 components, 50 Docker containers and 170 software packages total. The software development approach was eminently community-driven building upon the requirements elicited from the multidisciplinary scientific use cases that took part in the project. Leveraging the lessons learned from previous software development and e-infrastructure management projects [98], the project established a rigorous SQA process that stressed the fulfillment of a well-defined set of quality requirements from the very early stages of the software production. Thus, the

compliance of each code contribution was assessed enabled by the adoption of state-of-the-art software engineering practices. Chapter 6 delves deeply into the SQA process established during the lifespan of INDIGO.

DEEP Hybrid-DataCloud (DEEP)

The DEEP project's partnership was originated from the previous INDIGO consortium, but not in an all-inclusive approach, as the size of the project is considerably smaller. The project incorporated new members, but was mainly comprised of a selection of the leading members existing in this former project.

As its precursor, DEEP is a bounded software development effort that, unlike the INDIGO's vision of providing a generic cloud-based framework for better utilization and harnessing of the underlying resources, focused instead on the Machine Learning (ML) capabilities. The proposed architecture [99] is a distributed solution that covers the whole ML development cycle, including the "models creation, training, validation and testing to the models serving as a service, sharing and publication".

In terms of software production, the DEEP's SQA plan started off from the previous outcomes of the INDIGO era. DEEP project reused and extended the previous SQA process, with the particular purpose of extending the domains of action to the ML applications developed by the user communities, thus not only restricting those SQA benefits to the infrastructure-enabling services. Chapter 7 discusses the main insights obtained in the DEEP project with regards to the enhancement of the user experience.

EOSC-Synergy (SYNERGY)

Unlike the previously-described projects, SYNERGY is funded through the INFRAEOSC call to support the implementation of EOSC-relevant national initiatives. SYNERGY is an ongoing 3-year duration project that aims at expanding national e-Infrastructures and integrating thematic services, from nine European countries, in order to expand the avenues of research within the EOSC.

One key guiding principle embraced by SYNERGY is to ensure the accurate operation of the e-Infrastructure and thematic services being integrated by the project into the EOSC. The SQA-as-a-service (SQAaaS) solution, to be delivered as one of the fundamental outcomes of the project, is a clear proof of this fact. The SQAaaS will provide a means to automatically assess the software quality achievements through the –static and dynamic– analysis of the software. Hence, the SQAaaS is a definitive move towards defining the minimum quality attributes of the services being integrated in the EOSC. Chapter 9 will provide a comprehensive description of the architecture and implementation steps towards building the SQAaaS.

4.4 Conclusion

The ultimate goal of the EOSC is to deliver research-enabling services for multidisciplinary scientific communities. The way in which those services are delivered is key for building the required level of trustworthiness that fosters the progressive adoption by those communities, and eventually situates the EOSC as the reference point for doing research within the EU according to the Open Science standards.

The review of the integral documents released by the EC during the vision and implementation phases of the EOSC does not reflect the positive impact that developing software according to modern quality standards has on the final delivery of the services. Probably as a side effect of this, the ongoing EOSC-hub project, which is in charge of developing the initial federating core of services, is lacking of this software-oriented approach, relying on the inaccurate –as acknowledged by the project– TRL maturity measurement system to set the minimum quality criteria for the service onboarding.

The primary argument being elaborated throughout this chapter is that the EOSC should not only aspire to attain the federation and interoperability of the currently fragmented e-Infrastructures and RIs. Indeed, the EOSC must be built upon the beneficial outcomes and knowledge base gathered as a result of

delivering services to researchers during the course of numerous e-Infrastructure development and management projects, which took place before and during the lifespan of the Horizon 2020 programme.

In the early days of the European e-Infrastructure development, the focus was essentially –and erroneously– set on extending the capabilities and services built on the underlying distributed computing technology, based on poor software engineering practices. The resultant operational instability made apparent the need for a better balance with regards to the quality, especially in terms of reliability, of the services being offered to researchers. Since then, the increasing awareness of the importance of adhering to software quality procedures is tangible, as it will be shown throughout the next chapters.

This expertise shall be the groundwork for the delivery of services within the EOSC, placing value on the definition of an accurate criteria for the underlying software, and thus, avoiding the current vagueness that exists when estimating the maturity of those services.

A Story of Three Acts

A Story of Three Acts

Until now, the virtues of both the Open Science paradigm and the definitive role that the quality of software plays in its realisation have been discussed as part of the first block. The last chapter of this block, contextualized the study within the European frontiers, reviewing the vision and the steps already taken in the European Open Science Cloud (EOSC) implementation.

The previous Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud" showcased several challenges in the process of delivering services that are continuously being integrated into the EOSC, and those boil down to an imprecise estimation and assessment of the service quality and maturity. According to the hypothesis of this work, already introduced in the Section "Thesis Statement", the software is the enabler of the services, and thus, it has to be considered as a fundamental part of the EOSC implementation. Based on this assumption, this second block of the thesis delves into the solution proposed, which contributes to bolster the task of assessing the quality of those services.

Methodology followed in the next chapters

The following chapters provide an incremental and comprehensive application of a Software Quality Assurance (SQA) process, divided in three acts, from the definition to its practical implementation, complemented by the development of a solution for disseminating the culture built around it. The field of study are the European e-Infrastructures that, as stated in the previous chapter, have

collected expertise in developing and operating research-enabling services, as it is the case of the EOSC.

The *definition* act encompasses the identification of the set of suitable criteria, clustered around the so-called SQA baseline, that have a high impact on the quality of the software being produced, thus being as pragmatic as possible. The SQA baseline provides a solution to the guidance gap existing in the EOSC, enabling a more accurate estimation of the quality and maturity of the onboarding services.

Secondly, in accordance with the SQA baseline criteria, the *implementation* act describe the methodological and technological solutions to build a DevOps culture, underpinned by the tools provided by Chapter 3 "Building a Culture of Software Quality". Real examples are covered throughout the chapters in this act, which cover the perspectives and twinning of both the research projects that develop the software solutions and the ones that operate the e-Infrastructure through the use of such solutions. The aggregate of SQA and automation enables the success of this collaboration, manifested by increasing levels of stability and reliability within the e-Infrastructures. Similarly, the implementation of a DevOps approach within the EOSC context would improve the trust of the researchers in the services being offered.

Lastly, based on the promising results obtained from the practical implementation of the software quality criteria, the last act broadens the horizons beyond the e-Infrastructure level, outlining the architecture of general-purpose solution to assess the quality of the research software. The goal in this act is to convey and disseminate the SQA culture within the whole research software community, and how this culture impacts the implementation of the Open Science values in the EOSC services.

Act I

Laying out the Groundwork: The Definition of a Baseline for Software Quality Assurance

5

Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software

Part of this chapter has been published as:

Pablo Orviz Fernández, Álvaro López García, Doina Cristina Duma, Mario David, Jorge Gomes, and Giacinto Donvito. "A set of common software quality assurance baseline criteria for research projects". In: (2017). URL: *http://hdl.handle.net/10261/160086*

This document devises a pragmatic approach for the realization of quality software in academic research. To this end, we present a detailed discussion of

the essential criteria that are expected to find in the software products resultant from the research work. Hence, the present compilation is purposely oriented to match Open Science standards while being eligible to conduct the software development processes within research infrastructures, which was its initial *raison d'être*. Consequently, state-of-the-art methodologies in software engineering are at the heart of this baseline.

The insight collected in this work is aligned and elaborates on the guidelines included as part of the version 3 of the ***A set of Common Software Quality Assurance Baseline Criteria for Research Projects*** document [100], hereinafter referred to as the ***Software Quality Assurance (SQA) baseline***.

5.1 Background

Back in 2015, while in the starting phase of the INDIGO-DataCloud (INDIGO) [101], there appeared the need to compile a set of quality conventions that guided the planned developments. The project aimed at developing a solution comprised of a set of software that had varying backgrounds, ranging from more mature solutions –some having their own quality procedures– to brand new products that were required by the project's architecture design. In addition, contributions to major open–source projects were planned in the project's description of work, thus upstream acceptance became a key indicator for the project success.

As a result of the heterogeneous ecosystem that composed the project's architecture –different level of maturity, wide range of programming languages–, the focus was put in converging into a minimum viable, but generic enough, set of requirements. These requirements were complemented with additional good practices and recommendations acquired from different sources, such as software standards and engineering methodologies, industry–proven insights and processes implemented in leading open–source software projects. The foundational document [102] emerged as a project deliverable, setting the version 0 of the SQA baseline.

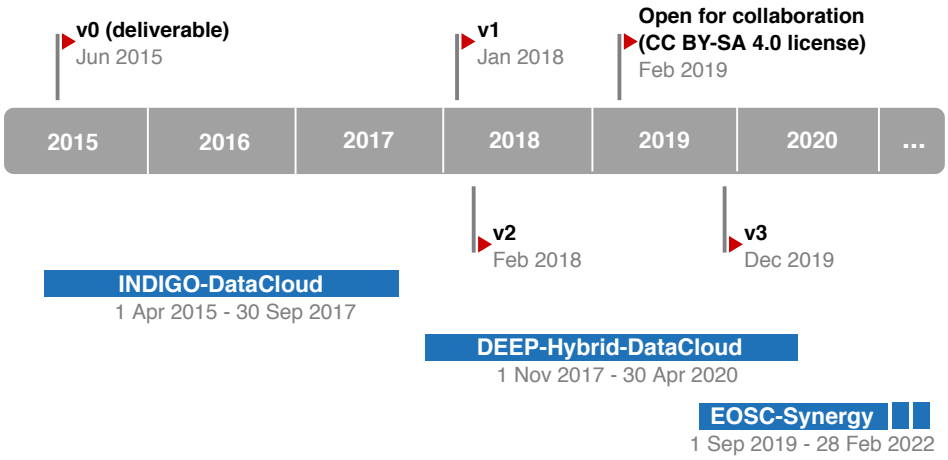

Figure 5.1: Representative milestones within the SQA baseline's roadmap. The associated Horizon 2020 e-Infrastructure projects that supported the implementation of the SQA baseline are highlighted at the bottom of figure.

All along the 30–month duration of the project the SQA baseline drove the development of more than 30 software components –scattered over around 200 code repositories–, and was gradually enhanced according to the emerging needs and new requirements. Far from being abandoned, the maintenance of the SQA baseline continued once INDIGO reached its end. The approval of two succeeding –but independent– software development projects, DEEP Hybrid-DataCloud (DEEP) [103] and eXtreme-DataCloud (XDC) [104], not only enabled the continuation of the SQA activities but extended their recognition and application to a broader public, as new partners joined in the respective projects.

In 2017, shortly after kick-off activities of both projects, the SQA baseline was compiled and published in a standalone document [100], meant to evolve on its own, with no further attachment to any particular project, but intended for general guidance on software engineering research matters. This was actually the turning point that eventually lead to the opening of the SQA baseline to external collaboration early in 2019 [105]. Finally, the last version (v3) of the

baseline was published in late 2019. Figure 5.1 illustrates the evolution followed by the SQA baseline since its inception.

5.2 Motivation

From a research e-Infrastructure perspective

As it can be extracted from the enumeration of the European initiatives described in Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud", the ***knowledge transfer*** between successive software development projects existed –by means of deliverables and milestones– , but it ***was not inclusive enough***. Unlike the long-term support of research infrastructures, these projects were short-lived, funded during a very specific period of time, and consequently, there was a natural discontinuity in the research and maintenance of the implemented processes.

The new forthcoming projects did not, nonetheless, start from scratch. They considered the working practices from the available documentation of the past activities, but eventually, the resultant knowledge base was not constructed in a way that it could be preserved and transmitted. Besides, this knowledge was ***reduced to the siloed expertise of the project consortium***, so no external feedback was usually considered, other than the literature on the matter.

Consequently, there were no comprehensive reference point for new projects to fall back on, in terms of developing quality software for the research infrastructures. This situation also affects the ***sustainability of flagship assets resultant from concluded projects***, where developers are left alone in the task of maintaining a successful software product. By no means software maintenance is an easy task, and should not be left on its own, evolving with no explicit guidance on the most basic quality features. Assuring software quality is key for its adequate maintenance and sustainability, as otherwise it would be hard or unlikely to make it grow and/or adapted to changing conditions such as the environment and/or the user requirements.

From a research software perspective

Research software development and maintenance often suffers from a notable ***absence of quality assurance realization***. As discussed in Chapter 2 "The Role of Software in Open Science", inherent factors are usually at the root of this reality. The SQA baseline presented hereinafter is an effective tool for reacting to this lack of awareness of software engineering practices in those academic environments.

5.3 Essential criteria for quality research software

The quality criteria are identified by virtue of their importance, either falling into the *Requirement* or *Good practice* category, and clustered around four key categories, each one discussed in a different section. The criteria within each of those categories meet the demands of the quality characteristics identified in Chapter 3 "Building a Culture of Software Quality" in order to establish the SQA process. Figure 5.2 shows the relation between those characteristics and the five categories from the SQA baseline.

Note that the original criteria, as outlined in [100], set the levels of criticality using the RFC 2119 convention [106]. For the sake of clarity, the ***criteria described herein relies instead on the requirement or good-practice*** binary form.

5.3.1 Code management

Requirement I: Create versions of the source code

Version Control Systems (VCSs), also known as Source Code Managers (SCMs), provide a highly flexible means to maintain a repository of content, perfectly suited for source code and/or documentation. The benefits of versioning your code are widely covered in software engineering literature, which include ex-

Figure 5.2: Mapping between the software characteristics (from Chapter 3 "Building a Culture of Software Quality") and the SQA baseline criteria's categories

tended capabilities for collaborative work in geographically dispersed environments, the ability to undo or revert changes, ownership management or backup [107, 108]. As in the previous code accessibility practices, reproducibility is again a particular area of focus, as code versioning –in conjunction with with data provenance– plays a decisive role in its improvement [109].

Good practice I: Maintain a clean history of changes

In code versioning terminology, a *commit* operation is performed for each significant change –feature or fix– in the code. As a result of a successful commit, an entry in the history is registered, characterized by a title and, optionally, a more elaborated description of the change. At a later stage, the VCS user will rely on this metadata to identify the changes that occurred at that time, therefore the commit operation plays a definitive role in maintaining a readable history.

To this end, the ***determination of when doing the commit should be driven by the identification of an irreducible improvement in the code***. This means that the developer must recognize the minimal set of modifications

–in a file or set of files– that accomplish a common goal. ***In what regards to the what, one–line, short and meaningful subject or title messages*** are crucial to facilitate forthcoming checks on the history. As a general rule, commit messages such as "bug fix" and "minor change" shall be avoided. Future successful retrieve, compare or revert operations will rely upon human-readable commit histories.

Good practice II: Use a branching strategy to separate your development and production versions

A more advanced feature of VCS is branching. The initial repository setup includes a default branch from where others can derive, in a tree-like structure. Conceptually, one could see a ***branch as a diversion from the original work***, useful for implementing new changes without affecting the regular development within the parent branch. The lifespan of a branch varies according to its purpose, following a parallel progress until the developer decides whether to consolidate it –through a *merge* operation– or keep it separate as a long–term version of the software.

SCM branch-based methodologies, such as `git-flow` [1] [110], advocate for the ***use of two long-term branches: development and production***. The development branch contains the new features and fixes that will take part in the next release. Accordingly, every change is only merged in the development branch, leaving the production branch untouched. At release time, the development branch gets eventually merged into the production branch. At this point in time, the release is *tagged* –following the naming convention of "Good practice VI: Use semantic versioning for your releases"–, which records in the SCM history the exact commit where both branches were merged.

According to this methodology, all the -smaller– new changes are tracked in individual and short-lived branches that, once approved, are meant to be merged

[1] Even though `git` tool is explicitly referenced here, the theoretical basis behind this workflow builds on the branching capabilities of VCSs, and thus, it is applicable to any VCS solution.

in the development branch. There is only one exception to this, the emergency fixes. They address security flaws, which are also tracked in separated branches, but merged in both the production and development long-term branches.

Good practice III: Maintain your long–term support versions

According to the complexity and/or requirements of the software project, additional long-term branches may exist in addition to development and production branches. ***Support branches are used to maintain multiple Long Term Support (LTS) versions*** and stem from the release tag that identifies the particular release to be supported.

As a result, ***support branches are not intended to be merged back to production***, but rather to be simply removed when reaching the expiration date. Unlike the production and development long-term branches, the ***support branches do only accept bug fixes***, not new features.

Good practice IV: Use an unambiguous naming convention for branches

Semantics are important when naming branches, especially in branching models as the one described in "Good practice II: Use a branching strategy to separate your development and production versions". Regardless of the SCM tool, a good practice is to ***fix a naming convention for discriminating among the different types of changes***. Thereby, a feature branch could be prefixed by the `feature/` identifier, followed by the branch name. The same applies to bug fixes, `fix/` or `hotfix/` and support, `support/` branches.

Good practice V: Use issue tracking

Issue tracking provides software management capabilities and facilitates organised software development [2]. An ***issue can be seen as a reminder to per-***

[2]Issues are particularly important when relying on the agile frameworks described in Chapter 3 "Building a Culture of Software Quality". They are the fundamental elements for track-

form a specific task in the source code or documentation. Issues can roughly be categorized as enhancements, which lead to the implementation of new code features or documentation enhancements, or anomalies, commonly mapped to bugs or documentation typos.

Issues are of most value when tightly integrated with the source code. The practice of ***adding a reference to the issue/s that the commit is addressing –within the title or description– contributes to the composition of a fine-grained and comprehensive history of changes***. Pull requests, as described in Good practice VII: Use pull requests, can also add references to open issues.

In addition to internal development purposes, ***issues are the best means for supporting users***. Thus, issues opened by users provide valuable feedback for the developers as they can potentially point out to bad performance problems or uncover defects found in the software. Besides, user input can lead to the development of new features or better approaches for existing functionalities.

Good practice VI: Use semantic versioning for your releases

As it was described in "Good practice II: Use a branching strategy to separate your development and production versions", tags are use to indicate the point in time of a new release. ***Tagging a release implies figuring out a version number that will identify it***. This version number should not be randomly set, but instead, it should follow an incremental approach and, additionally, give meaningful information to the users of the software about the magnitude and significance of the changes contained therein.

To meet this goal, the ***Semantic Versioning (SemVer) specification [111] proposes version numbers in the form `X.Y.Z`, made up of non-negative incremental numbers***. In order to showcase the dimension and backward compatibility of the changes introduced, the fields within `X.Y.Z` correspond, respectively, to a *major*, *minor* and *patch* version. Therefore, an

ing work inside the Kanban boards or Scrum sprints.

increase in the major field represents a backwards incompatible version, whereas increases in minor or patch versions imply, respectively, compatible new functionalities or bug fixes with regards to the immediately preceding release.

5.3.2 Collaborative coding

Requirement II: Use public forges to distribute your work

Public forges do not only provide code hosting services but also build on the value of transparency in order to offer *social coding* capabilities. GitHub [16] is today's de-facto social coding platform, widely used for academic research software, and is increasingly attracting the attention of recent studies about its influence on software development practices [112].

In the research software context, the power of social media is ***boosting the capacity to learn from others –education–, thriving community building –collaboration– and scientific recognition –reputation–*** at a significantly larger scale [113]. It is also playing a key role in ***supporting the sustainability or maintainability of the software***, as a result of providing mechanisms to support *forking* open source code [114].

Furthermore, ***public forges provide an extra means of code preservation***. Code preservation is a prerequisite of an accurate identification of software, otherwise it cannot be discovered, accessed, cited or reproduced. Unlike the traditional belief that considered executables or binaries as the software products to preserve, the appearance of portability issues when trying to reuse old programs in new hardware changed the focus towards the underlying source code. Indeed, the source code represents a valuable and culturally rich object that exposes the creativity and knowledge of the programmer. Source code, unlike binaries, can be read, extended and sustained over time.

Nevertheless, it should be noted that code hosting platforms are driven by commercial interests that may lead to the potential risk of suffering one-sided policy changes that directly affect the long-term availability of the code [115]. Yet, even skeptics know that nowadays these platforms offer the greatest value

SQA criteria	Expected outcome
Code management	
Requirement I: Create versions of the source code	SCM tool usage (e.g. `git`)
Good practice I: Maintain a clean history of changes	Atomical & topical commits
Good practice II: Use a branching strategy to separate your development and production versions	Short (feature, fix) & long-term (`master`, `develop`) branches
Good practice III: Maintain your long–term support versions	Individual long-term branch for each LTS version
Good practice IV: Use an unambiguous naming convention for branches	`fix/` for bug or emergency fixes `feature/` for enhancements `support/` for LTS versions `release/` for tracking a release
Good practice V: Use issue tracking	Internal development issues (enhancements, bugs) Support issues (incidence, user wishlist)
Good practice VI: Use semantic versioning for your releases	`x.y.z` release versions
Collaborative coding	
Requirement II: Use public forges to distribute your work	Publicly accessible code repository with social coding capabilities (e.g. GitHub)
Requirement III: Make clear your contribution policy	Contribution guidelines & acceptance criteria in a `CONTRIBUTING` or `CONTRIBUTION` file
Good practice VII: Use pull requests	Improved code review Unique means of adding contributions to `master` Make reference to open issues
Good practice VIII: Protect your long-term branches from direct modifications	Safeguard production (`master`) version from direct pushes
Code accessibility	
Requirement IV: Make your code open and publicly available	Free/Libre and Open Source Software (FLOSS)-compliant license in `LICENSE` file Copyright header in all the source code files
Requirement V: Make your software findable, reproducible and citable	Digital Object Identifier (DOI)

Table 5.1: Code management, accessibility and collaborative coding criteria.

to manage and share the code, especially when sustained funding is a common issue in science, which hinders the long-term preservation of scientific code and data in a non-commercial scene.

A recent archiving effort conducted by a non-profit organization, known as Software Heritage [61], is conducting the ambitious task of archiving all the open source code available in the currently existing worldwide software forges. Consequently, ***the simple act of using an FLOSS license –see Requirement IV: Make your code open and publicly available– and making your source code available in any of the currently popular public forges –see requirement 5.3.2–, will eventually guarantee the preservation of your code***.

Requirement III: Make clear your contribution policy

As it have been already discussed, using public repositories opens up new opportunities of external collaboration. Clear guidelines to engage potential contributors [112] are required in order to describe the contribution process. Those guidelines are recommended to be ***compiled in a `CONTRIBUTING` file at the root of the code repository***.

While the content of this file is not fixed, it should at least ***state clearly the acceptance criteria*** –e.g. under a Governance section– so external contributors are aware of how decisions are made and by whom.

Good practice VII: Use pull requests

Pull Requests (PRs) represent the cornerstone of software collaboration in social coding environments. They are ***used to tackle any new contribution, whether it is done to personal or external code repositories***. In the latter case, a PR implies *forking* or duplicating the relevant version of the target software repository to one's work-space in order to commit the required changes. The PR is then created back in the original project, thus notifying the repository owners about your intention of including the proposed changes into

the codebase.

According to social metrics, PRs that address a particular issue thoroughly and efficiently are readily accepted, whereas the ones suggesting a large change are less likely to go through [112]. Therefore, the contribution policy from "Requirement III: Make clear your contribution policy" should describe how PRs are expected, and thus, avoid undesired inputs.

Good practice VIII: Protect your long-term branches from direct modifications

Despite the benefits of maintaining a stable and workable production versions that the branching strategies provide, most of the code contributions still come in the form of direct code commits to the main or production branch [112]. Even though it is a faster way of adding changes, it is also a riskier approach as commits containing errors will make the production version malfunction. Branch protection capability is provided by the code hosting platform, not by the SCM tool itself.

5.3.3 Code accessibility

Requirement IV: Make your code open and publicly available

Software is exclusive copyright by default [3], so it is imperative to clearly state the license agreement the software is adhered to, even in the case that the software shall be eventually copyrighted. Otherwise, software is left in a situation of legal uncertainty with regards to its accessibility and distribution [117] that may discourage interested parties from using it, as they may incur into intellectual property violations.

The essential step for declaring your software as open is to adhere to a non-propietary license, and in particular, a ***license that is compatible with both the Free/Libre Software Foundation and the Open Source Initiative***

[3] Meaning that interested parties are not allowed to use, modify or share your work [116]

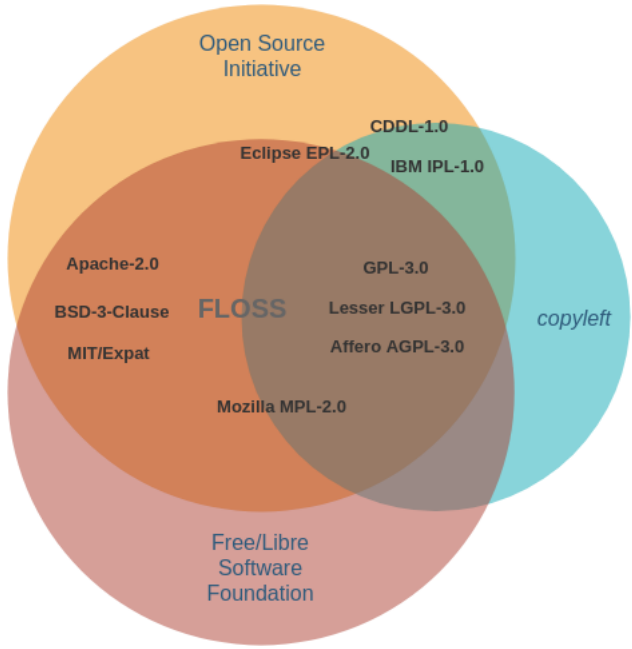

Figure 5.3: Popular open source licenses used for distributing the software, where the group of FLOSS-compatible licenses are located within the darker orange area. Additional *copyleft* –used to preserve the openness of derivative works– information is provided, either in limited (MPL-2.0, EPL-2.0, CDDL-1.0) or full compliance (IPL-1.0, GPL-3.0)

requirements. They do present philosophical and political discrepancies among them [118], but in practical terms they converge to endorse a similar set of licenses, as it can be seen in Figure 5.3.

In order to express the licensing adherence, the convention is to ***add a `LICENSE` file alongside the codebase***, containing the explicit definitions of the selected software license. This action shall be required for any active software project.

Additionally, the `LICENSE` file shall be complemented by ***adding a header – containing both the copyright line and the link to where the full license***

declaration is found– at the beginning of each source file existing in the code base. Thus, not only the licensing notice is more accessible than in a separate file, but most importantly, the legal implications of using your work are effectively stated [4].

Requirement V: Make your software findable, reproducible and citable

The FORCE11 Software Citation Principles document [120] collect the essential requirements on how software should be cited. Here, the ***unique identification of software is recommended to be done through the use of Persistent Identifiers (PIDs)***, granted that they are the current standard for digital products such journal publications. Thus, PIDs should facilitate the access to the software itself, and the metadata should at least provide the minimum required information intended for its accessibility.

Multiple metadata schemes do exist nowadays for different communities, each one having their own specifications. Fortunately, the CodeMeta initiative [121] is facilitating their convergence by creating a minimal schema for describing scientific software, aiming at achieving the successful exchange of metadata among the most popular software repositories and organizations. This way, the ***metadata is placed in the repository of code as a JavaScript Object Notation (JSON) file, following the specification of the CodeMeta-2.0***, filling in the information relevant for academic credit or citation, replication or reproducibility details –such as version or required dependencies– and findability or discoverability –by e.g. using keywords–.

Once having the tools for uniquely identifying the software, ***academic software developers should follow the path of journal publishing***. There are specific journal for publishing software, such as SoftwareX [122] and the Journal of Open Source Software (JOSS) [123]. The latter is tightly aligned with the described software citation principles as it issues PIDs and allows the publication

[4] As an example, OpenStack's developer guidelines include the presence of the license headers as a requirement [119]

of multiple minor or major versions of a piece of software, as long as it complies with JOSS code of conduct [5].

5.3.4 Verification and validation

Good practice IX: Adopt agile principles for managing your software project

Agile development methodology suits software projects of all sizes, but, according to the experiences that will be described in the next chapters, it has proven to be a particularly valuable tool for managing the development and integration stages when multiple software components come into play.

When eliciting the user requirements, two different types of requirements need to be considered: functional and non-functional. ***Functional requirements are gathered from user feedback, while non-functional are part of the software's quality performance***. Whereas non-functional requirements do not need any further processing, as they are carried out according to existing performance specifications, the requirements coming from the users undergo a refinement process that results in the breakdown of a series of technical developments. According to the agile methodology, this process shall be iterative, thus flexible enough to adapt to changing requirements throughout the entire software development life cycle.

When running the project development, requirements are best handled through issues –see "Good practice V: Use issue tracking"– and disposed in accordance with the agile framework in use, either Kanban or Scrum, according to the flexibility needed to release the final or intermediate [6] product. Agile frameworks have become the standards for software development management [7]

[5] `https://github.com/openjournals/joss/blob/master/CODE_OF_CONDUCT.md`

[6] or Minimum Viable Product (MVP), which was discussed in Chapter 3 "Building a Culture of Software Quality"

[7] GitHub has built-in capabilities for Kanban frameworks.

SQA criteria	Aut	Good for	Means
Verification and Validation (V&V)			
Requirement VI: Test the individual units of the code	✓	Code reuse Early identification of bugs Self-documentation, facilitates integration	Testing libraries Mocking objects
Requirement VII: Address functional requirements	✓	Meet technical requirements Address functional accuracy	Testing libraries Mocking objects
Requirement VIII: Check the level of integration and interconnection with coupled components	✓	Software flexibility within uncontrolled environments, Interoperability	Testing libraries, IaC container-based frameworks
Requirement IX: Ensure new changes do not jeopardize the operation of software's existing features	✓	Software consistency in the presence of new changes	Re-running unit/functional/integration tests
Good Practice IX: Supplement functional requirements with behavioural testing	✓	Validate user stories	`Given-When-Then` pattern
Requirement X: Adhere your code to a code style standard	✓	Code readability	Analytical tools for style standard compliance (linters)
Requirement XI: Assess the security on your software	✓	Uncover code security flaws	Static Application Security Testing (SAST) (code linters, dependency analysis checkers and secure coding)
	✓	Protection against vulnerability exploitation	Dynamic Application Security Testing (DAST) (vulnerability scanning tools)
Requirement XII: Broadening the perspective with peer reviews of the code	✗	Detect the suboptimal aspects of a code change	Code-review tools, PRs

Table 5.2: V&V criteria in the SQA baseline.

Requirement VI: Test the individual units of the code

When writing unit tests, programmers commonly rely on specific libraries –available for the programming language in use– that facilitate to a large extent their implementation, such as by *mocking* –or simulating the behaviour of– the objects used in the unit that are irrelevant for its evaluation. These libraries seamlessly execute the set of test cases defined for each unit that, combined with the code coverage tools, provide the percentage of code being executed over the total number of code statements, conditionals and/or functions. ***Test cases are best written alongside the implementation of the relevant code***, so that coverage does not drop dramatically on high-intensive periods of coding.

The coverage value gives a good estimate of the consistency of our code. Nevertheless, as with any other type of testing, high coverage values yield to upper costs in terms of effort. The benefit obtained is not necessarily balanced with effort, tending in fact to follow a non-linear pattern. According to our experience, and supported by the literature on this matter [124], a ***coverage value that approximates to 70% represents a good compromise between testing benefits and dedicated effort***.

Requirement VII: Address functional requirements

As far as where to focus the testing effort, it is a fact that not all the sections of the code are of equal importance. The Pareto Principle or 80/20 rule [125] states that a 80% of a program usage is handled by a 20% of the code. Accordingly, the job of the programmer is to identify the sections where bugs are most likely to lead to negative effects to end users, and focus on rising their coverage. Note that this is true also for the case of the unit tests. Similarly to those, the test cases for the functional tests shall be written at the time of adding each new functionality.

Requirement VIII: Check the level of integration and interconnection with coupled components

When writing integration tests, one should always assess the viability of automation, although this possibility is tightly coupled to the particularities and complexities of the components and interfaces involved. It is indeed a kind of testing strategy more likely to rely on manual intervention. However, with the spread of container virtualization technologies and orchestration tools, the composition of complex setups, using simple definitions in configuration files, has been largely facilitated, and consequently the automated execution in continuous integration environments.

Running integration tests is estimated to be more expensive in terms of effort and resource consumption than the aforementioned unit and functional tests, particularly higher when lacking of automation. Thereby, integration tests are not required to be triggered for minor changes in the code. As a rule of thumb, integration testing is expected when releasing a new minor or major version of the software, thus excluding regular changes in the source code (pull requests) or patch releases.

Non-functional or performance requirements might be assessed along integration testing. Scalability, usability, volume or load tests fall into the non-functional testing category. Based on their resource-demanding characteristics, non-functional tests shall be carried out through automated means.

Requirement IX: Ensure new changes do not jeopardize the operation of software's existing features

Regression tests should be those –from the available units, functionalities and integration test cases– that maximize the coverage of the most representative characteristics of the software. The Pareto principle described in "Requirement VII: Address functional requirements" provides a starting point to minimize the number –by determining the most relevant– regression test cases of the software [126].

Good Practice IX: Supplement functional requirements with behavioural testing

Behaviour-driven –or acceptance– tests are the result of driving a behaviour-driven development (BDD) methodology to address the user requirements. Hence, BDD links with the user stories themselves, so it appears as a higher level type of testing when compared with functional testing. Indeed, a BDD test is usually aligned with one or more functional tests.

As it has been discussed, functional testing is driven by the developers, but lacks of the user perspective. BDD tests should be conducted, and ideally developed, by the end users of the software or the SQA team. It is important to set the developer aside, since the goal is test the software according to the user expectations. Otherwise, the likelihood of a biased analysis increases.

Current approaches to BDD rely on the `Given-When-Then` pattern, which allows the user to express the diverse situations to be faced when interacting with the software. `Given` describes the state upon which the action identified by `When` will occur. `Then` validates the user story by defining how the behaviour of the system shall meet the user expectations.

Requirement X: Adhere your code to a code style standard

A code style is attached to a given programming language. Different style formulation efforts had attained broad community consensus and sustained maintenance, eventually considered ***de facto code style standards***. Conversely, custom or individual's agreed set of style rules, while commonplace in the past, should be avoided and only considered in the hypothetical event of programming languages without existing standards. The usage of community–driven standards are strongly encouraged as they are *well defined* –in terms of evidence on the value or contrasted suitability of each convention–, *supported* –having a strong community of experts behind driving the maintenance of the definitions and analytical tools– and *established* –used by the leading open–source projects–.

Practically speaking, the selection of a style standard is highly dependant on the availability of tools –i.e. *linters*– that automatically check the conformity of the source code with the conventions that comprise it, rather than being guided by the analytical superiority of one standard over the other. Usually there exists this duality between maintainability and consistency, so that the most accurate style definitions are in fact the best supported.

Requirement XI: Assess the security on your software

Security must be considered at all the stages of an application development life cycle. Software security assessment shall be at least guided by the aid of automated tools available for the static and dynamic analysis testing.

SAST applies to the code and needs to be part of the CI stack, thus regularly scheduled for each change in the code. The goal is to uncover the security flaws that might have been added in the last modifications of the code. A rather complete SAST assessment can be supported on the tripod of code linters, dependency analysis checkers and secure coding.

Linters provide capabilities for detecting bad practices in the code, such as file or assertion handling, untrusted connections to remote locations or usage of unsafe libraries or protocols. Dependency checks [8] focus instead in the presence of disclosed vulnerabilities –identified by its Common Vulnerability and Exposure (CVE) code– for the libraries or modules used by the software, whose security risk should not be under appreciated as they can represent up to 80% of the total code of an application [127].

Both types of automated SAST tools [9] are necessary complemented, and preceded, by the application of *secure practices* while coding. Guidelines such as the ones provided by the OWASP foundation [128] provide a convenient description of the common programming mistakes, how they are related to security

[8] As an indication, GitHub platform already performs dependency analysis checks on every hosted source code repository to increase the awareness amongst their users.

[9] The Open Web Application Security Project (OWASP) foundation maintains an exhaustive listing with the most relevant SAST tools available at `https://www.owasp.org/index.php/Source_Code_Analysis_Tools`

issues and the means of avoiding them. Needless to say that, in the event of accessibility to expert counseling, the three aforementioned practices are significantly improved by *secure code reviews*, as it is considered as the "single-most effective technique for identifying security flaws" [129].

An accurate DAST implementation should include protection against vulnerability exploitation through the vulnerability scanning tools [10] and penetration testing. While the latter is the most effective, it requires a solid background on security testing. Conversely, the vulnerability scanning tools do not require such background since it provides an automated means to look for common security vulnerabilities that can be exploited from the outside.

Requirement XII: Broadening the perspective with peer reviews of the code

Previous requirements and good practices stressed the importance of relying on automated programs to speed up –and increase the effectiveness of– the code verification process. Unlike them, ***code review is explicitly meant to be done manually***, so no automation is desired as, in this specific case, we are seeking a human perspective about the suitability of the change.

The various areas where source code reviews are particularly fruitful are summarized in Box 3.1. As code reviews' main focus is to address the suboptimal aspects of a source code change, comments can easily take a negative tone. Therefore, a neutral language should be used, avoiding personal or possessive pronouns, always seeking constructive criticism. Moreover, comments need to be concise, avoiding debate or excessive dialogue, and state unambiguously the actions expected from the author.

There are dedicated tools for carrying out code reviews. However, following the success of social coding platforms, PRs are the most used means of code review, based on their capacity to attract feedback from outside the project and

[10] Again, OWASP provides a benchmarking analysis of the most noteworthy vulnerability scanning tools currently in the market. See `https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools`

the fact of not having to deploy and maintain an additional tool.

Box 3.1: Source code reviews checklist

Goal or scope This is the first and most fundamental check done at code reviewing, which has to be imperatively performed by humans. Although the change has been correctly implemented, documented and tested, it may be providing an irrelevant feature or bug fix.

Test completeness . As code review stage represents the last step in the verification of a source code change, the results from SAST and DAST checks (this may not include the integration testing) should be already available. In this area, code reviewers should focus on the identification of uncovered units or sections of the code that directly impact the functionalities required by the end users [a].

Security assessment Code reviews can include an inherent security assessment of the risks that the candidate changes may introduce. The expected outcome in this area shall safeguard the security model of the software, ensuring that it has not been downgraded or compromised by those changes.

Compliance with best coding practices . Code reviews offer the opportunity to analyse the software quality conventions that cannot be otherwise assessed by automated mechanisms. In particular, reviewers should go through the checklist of good practices for code management, such as commit messages and history –see "Good practice I: Maintain a clean history of changes"–, branch naming –"Good practice IV: Use an unambiguous naming convention for branches"– or that referenced issues are completely covered by the changes introduced –"Good practice V: Use issue tracking"–.

[a] remember the Pareto Principle in "Requirement VII: Address functional requirements"

SQA criteria	Aut	Means	Outcome
Software uptake			
Requirement XIII: Comprehensive documentation	✓	Markup language	`README` (source code repository)
Requirement XIV: Treat documentation as code			Audience-specific documentation (doc repository)
Good practice X: Ease the deployment of your software	✓	Continuous Configuration Automation (CCA) tools (Ansible, Puppet)	CCA module (code repository)

Table 5.3: Software uptake criteria in the SQA baseline

5.3.5 Software uptake

Requirement XIII: Comprehensive documentation

Documenting a software product is often regarded as a tedious task, pictured by the well-known resistance of developers [130], since, unlike programming, it does not comprise creative work. The agile manifesto [55] advocates for the bare-minimum generation of documentation as stated by the "working software over comprehensive documentation" principle [11].

Nevertheless, if we observe the current trends, no relevant or successful software possesses deficient documentation, and indeed, it exists a direct relationship between the popularity of a project and the consistency of its documentation [112]. Hence, ***every effort made in order to maintain a comprehensive and readable documentation does pay off***.

Documentation is written to be read. Assuming that our source code is

[11] This statement raised broad debate and controversy within the software engineering community. S. Rakitin led the criticism to agile approach through his ironic statement "real programmers don't write documentation" [131].

readily available in a public forge, the often-neglected ***`README` file is the landmark of the software, and consequently, it shall contain the relevant information that facilitates the basic understanding about the purpose and scope of our software***. Figure 5.4 illustrates the expected content of a `README` file.

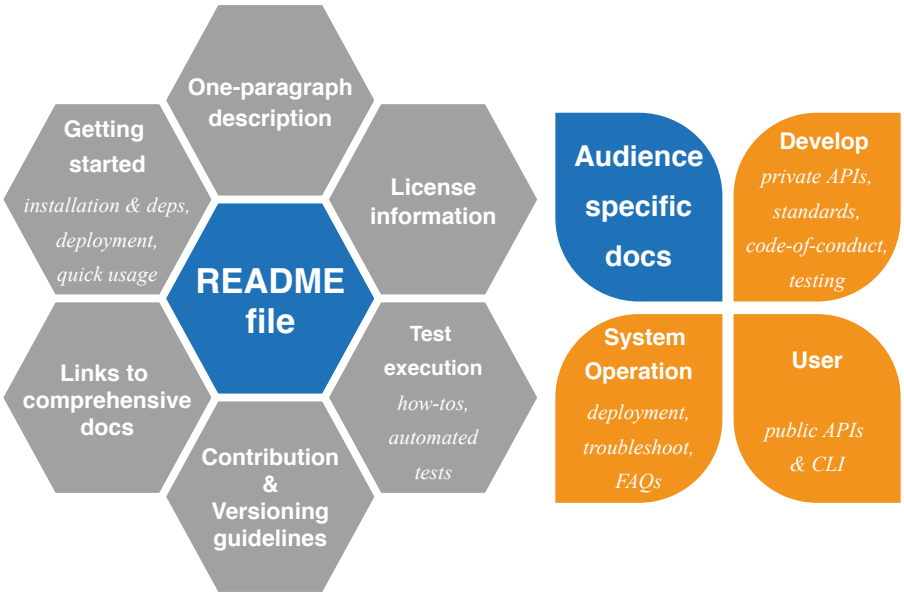

Figure 5.4: Documentation expected for any software project. Assuming that the code repository is publicly available, the `README` file content –figure on the left side– becomes essential. The more thematic and audience-specific documentation –right figure– is usually available in documentation-specific online repositories, so it is important to link them appropriately.

On the other hand, the more elaborated documents are the ones targeted to the specific audiences of our software. There are no fixed recommendations about the content since it highly depends upon the given software. A good practice is to improve the comprehensibility of our documentation is to put ourselves in the place of our audience's shoes, without taking the most obvious detail for

granted, no matter if dealing with technical or user-oriented documentation. Being diligent at this stage can contribute to software adoption.

The ***documentation shall be thematic according to the different needs of the final consumers, who are frequently comprised of end users, developers and operators***. Common requirements for these types of documentation are showed in Figure 5.4.

Requirement XIV: Treat documentation as code

Under the ***prerequisite of using of plain text*** format, the development of documentation can benefit from all the aforementioned good practices of the code. In combination with markup languages, such as the popular *Markdown* or *reStructuredText* solutions, plain text can be rendered in an appealing way to readers, so there is no real drawback with the results that can be obtained with rich text formats.

It is highly recommended to ***place the documentation next to the code, usually under a `docs/` directory*** within the repository of code. By acting this way, changes in the code that have associated modifications in the documentation can be then reviewed through the same PR, thus facilitating the code reviewer's endeavour. Likewise, corrective SCM operations, such as a *revert* or *amend*, can be done in a single step, without the burden of doing the same operation in two different repositories. Repositories hosting exclusively documentation are only advisable when in the need of managing documentation that is not attached to a specific software product.

Whilst code repositories provide the means for documentation composition and management, ***there are specific repositories [12] used for rendering navigable documentation***. They integrate seamlessly –via event notification– with those software forges, so that any given change in the documentation text files –within the `docs/` directory– that is merged in the production branch is automatically built and rendered in the associated documentation repository.

[12] See *Read the Docs* [132] or *Gitbook* [133]

Good practice X: Ease the deployment of your software

Ease of deployment is an indispensable requirement for software adoption. Unfortunately, not all the software out there is easily deployable, requiring cumbersome installation and configuration steps that, all too often, lead to frustrated attempts. According to the scope of the software, this unpleasant situation is either faced by the infrastructure operation teams, more skilled to deal with complex deployments, or directly operated by the end user, where this situation is most problematic.

While documentation plays an important role, the advent of ***CCA methodology shifted the complexity of deploying the software towards the developer of the code***. A CCA solution [13] provides a programmatic approach to a system's deployment using a high-level declarative language, so that it maintains a workable system with the minimal interaction of the end user or the infrastructure operator. Based on these facts, ***leveraging CCA is the most convenient way of raising the adoption of a software product***.

As it was the case of source code, and subsequently, the documentation, the development of a CCA module is best managed with a SCM tool. But, unlike the documentation, it is recommended that the code is managed in an individual code repository, separated from the main source code of the software. Thus, as a self-contained software product, it can be distributed to the official CCA-related repositories and re-used by the community.

On the other hand, the disruptive advent of the container technology [14] motivated the appearance of Infrastructure as Code (IaC) solutions that, in addition to the CCA tools, provided the means of not only deal with the software, but also provision the underlying infrastructure. As a consequence, container images are increasingly becoming a unit of deployment, whose script-like definitions are increasingly being added in software's code base.

[13] Most notable examples are Ansible [134], Puppet [135] or Chef [136]
[14] Docker [137] is the most prominent and widely-used container technology nowadays.

5.4 Conclusion

Producing quality software is by no means an easy task. Regardless of the size of the project, driving the software development life cycle always requires an *ab initio* definition of the essential criteria that will guide each modification in the underlying source.

Back to the inception of the SQA baseline in 2015, no such a compilation of minimal requirements, oriented to researchers and with a strong pragmatic approach was available to the extent of the current knowledge. Obviously the literature largely dives into the big complexities and capabilities of software engineering, but none provided a high-level, how-to-like view that covered all the required topics to aid the incipient projects contributing to the implementation of research infrastructures. Over time, the SQA baseline evolved towards promoting the values of a software quality culture, and thus, resulted also of great interest to individual scientists and communities, as the vast majority of the compiled know-how is generic enough to be applicable to any software development effort.

The SQA baseline guided the development of software within a series of research infrastructure development projects, ***establishing a clear path for real knowledge transfer***. Subsequent projects built on the outcomes of the precedent, without the need of reinventing the wheel. Having an open-to-collaboration and accessible document promotes self-maintenance –through a community endeavor– without the need of sustained or long-term funding. At the same time, the fact that any software expert or enthusiast appears as a potential contributor broadens the scope beyond the project partnership.

The use of standards is promoted throughout the SQA baseline. In particular, the code style requirement, either supported by a strong community or being the de-facto standard for the programming language in use, enables –either received or sent– upstream contributions. In conjunction with the required types of testing, any ***SQA baseline-compliant software project is in the best disposition to successfully contribute to any external project***,

regardless of their contribution policies.

In regards to the software engineering methodologies, the SQA baseline put the focus on acting right at the start of the software life-cycle, by promoting the ***evaluation of each modification*** done in the codebase, which appears as one of its distinctive features. As it will be thoroughly discussed in the production implementation of the Act Two, implementing such a scenario would not be realistic for the economics of software without the ***adoption of automation***. Although not implicitly mentioned in the baseline, the adherence to DevOps principles are active in the background.

Software manufactured following the SQA criteria is readily *accessible and re-usable* by other researchers, meaning that the code is encouraged to be written with *long-term sustainability* in mind. Improving the *reliability* of the software is achieved through the collection of noteworthy testing strategies that put the focus on meeting the requirements of end users, thus balancing out the invested time, cost and effort.

As pointed out in Chapter 2 "The Role of Software in Open Science", one of the reasons to conclude that software was not equally treated as data is the fact that there is no formal policy or research regulation about software. In this chapter, the SQA baseline defines the criteria for accurate software development and maintenance, and accordingly, serves the purpose of a regulation for quality software in research.

Act II

Where Theory Meets Praxis: the Implementation Process

6

Developing quality software from its origin: the INDIGO-DataCloud project

Part of this chapter has been published as:

Pablo Orviz Fernández, Mario David, Doina Cristina Duma, Elisabetta Ronchieri, Jorge Gomes, and Davide Salomoni. "Software Quality Assurance in INDIGO-DataCloud project: a converging evolution of software engineering practices to support European Research e-Infrastructures". In:* Journal of Grid Computing *18.1 (2020), pages 81–98. DOI: 10.1007/s10723-020-09509-z

The INDIGO-DataCloud (INDIGO) project featured the first implementation of the Software Quality Assurance (SQA) baseline, whose initial formulation was originated as part of the project design. The project's SQA process laid out the groundwork for foregoing projects, such as DEEP Hybrid-DataCloud

(DEEP) and eXtreme-DataCloud, especially in relation to the insights obtained in the use of automation for the software Verification and Validation (V&V) processes. As a consequence, a series of requirements from the previously discussed SQA baseline can now be automatically evaluated through the implementation of the DevOps approaches –depicted in Chapter 3 "Building a Culture of Software Quality"–, thus enabling the consolidation of the change-based approach advocated by the baseline.

6.1 The Software Quality Assurance process

Right from the start, INDIGO placed great value on control mechanisms for software development, allocating considerable effort in this direction. As a result, novel software engineering approaches were adopted in order to address the new challenges that were planned throughout the course of the project. One of those challenges was to achieve a seamless integration with the quality procedures of external open-source platforms that the INDIGO aimed at contributing. In most cases, those platforms, notably OpenStack [138], follow very well-defined quality procedures, and consequently, proactive measures had to be taken within INDIGO in order to be ready for prospective contributions.

The INDIGO roadmap for attaining quality in the software being produced is condensed in Box 1.1. Following the description of the SQA baseline, this chapter focuses in the third action, i.e. the automated implementation of the software development and release management phases leveraging DevOps practices. The insights obtained will be underpinned throughout the text by concrete results and metrics.

Box 1.1: INDIGO SQA process roadmap

1. The initial formulation of the *SQA baseline*, previously covered in Chapter 5, set the ground for the subsequent actions.
2. The definition of a set of *quality metrics*, based on the ISO/IEC 25022:2016

standard [47], to monitor the development, release and maintenance phases. The metrics were obtained programmatically from several sources, such as GitHub API [139] and the project's Jenkins Continuous Integration (CI) service.

3. The promotion of *automation* to enable the per-change based assessment promoted by the SQA baseline, aiming at accelerating the delivery process of new versions of the software.
4. A post-release and pre-production validation to be carried out using the project's *testbeds*, where the performance and integration between the developed components were evaluated. Once the integration was successfully tested, the software was deployed in *production environments* using the European Grid Infrastructure (EGI) e-Infrastructure.

6.2 Software verification, validation and delivery through DevOps

The amount of requirements compiled in the SQA baseline is high enough to justify by itself the need of automation. But in addition, automation is crucial for granting the compliance of those requirements for every code modification, regardless of its size.

In INDIGO, automation was achieved by leveraging on Continuous Integration and Delivery (CI/CD) services, in particular, the Jenkins CI service [74], introduced in Chapter 3 "Building a Culture of Software Quality". The outcome is the materialisation of CI/CD *pipelines*, as depicted in Figure 6.1, that comprise the diverse stages that any change has to go through in order to be successfully delivered in production.

As discussed in Chapter 3 "Building a Culture of Software Quality", the automation of the V&V processes increases the overall reliability of the produced software: automated testing is more time-efficient, leading to a higher code coverage and increased defect detection. To put into numbers, during the lifetime of INDIGO, the total number of defects detected in the V&V stage

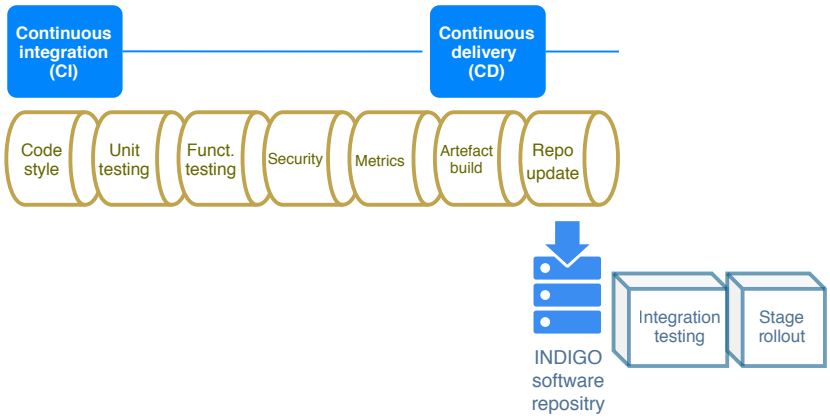

Figure 6.1: A simplified version of the delivery process implemented in INDIGO. The CI/CD pipeline, represented in dark orange, carries out –automatically– the V&V processes and delivers the generated software artefacts into the repositories. These are used to perform the subsequent –manual– integration testing and validation on production environments, through the stage rollout.

–that corresponds to the CI phase in Figure 6.1– surpassed by a factor of 30 the ones reported by external users. Figure 6.2 shows the evolution of bug detection throughout the project.

Technical implementation

Figure 6.3 illustrates the INDIGO CI/CD solution. Apart from the aforementioned Jenkins CI service, the implementation leveraged other technologies, such as GitHub, for the code management, and Docker, both for resource provisioning and to deliver container artefacts.

The project defined a source code contribution workflow based on GitHub PRs, according to the SQA baseline's Good practice VII: Use pull requests. Upon code modification, GitHub sends a notification to Jenkins that triggers the execution of the CI/CD pipeline, going through all the stages represented in the previous Figure 6.1.

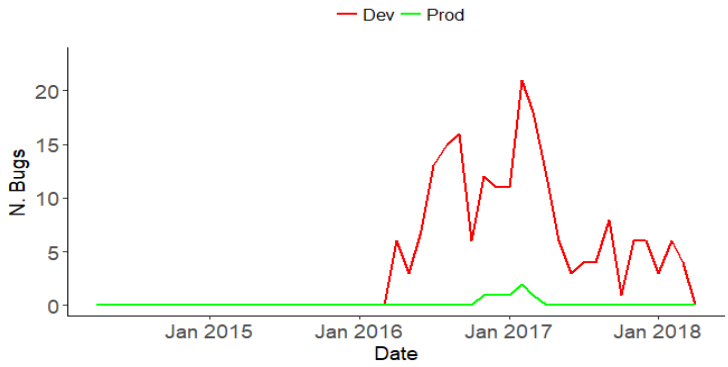

Figure 6.2: Comparison of the number of software bugs – documentation typos are excluded – detected during development and production or stage rollout phases, over the lifetime of INDIGO project. Whilst production-related bugs correspond to software bugs filed by users of the EGI e-Infrastructure, the development defects stem from CI/CD and integration stages (source: GitHub issue tracking). Note that this chart does not show all the bugs detected by the execution of the CI/CD pipelines since they are usually fixed by developers on the fly without being tracked through issues.

As Figure 6.3 shows, the automatic upload of artefacts directly in the production branch is intentionally avoided as a precautionary measure, and accordingly, only a *preview* version of the artefact is uploaded to the repositories. This preview release is subsequently used in the integration phase and, which in case of passing the tests successfully, it becomes the production version.

Once the CI/CD pipeline execution ends, either because it ended successfully or it failed on the way, Jenkins notifies back to GitHub the exit status, so it can display the appropriate result. As each change is verified and validated, the chances of early detection of defects increase. Within this scenario, the cost of defect solving is dramatically reduced and the reliability of the software solutions improved, as any bug or design issue is likely to be detected and subsequently corrected, all of which performed at this phase.

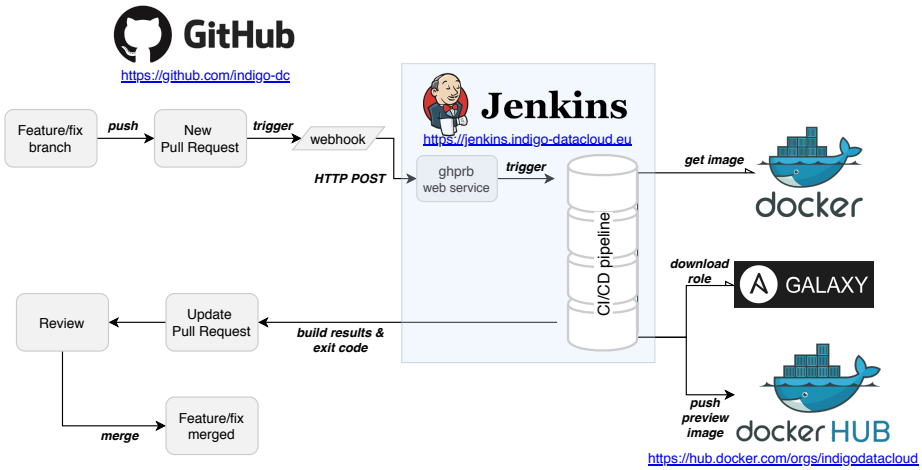

Figure 6.3: Implementation of the CI/CD workflow for the INDIGO core components. The Pull Request (PR)-based code workflow facilitates the V&V of incoming features and fixes. The webhook residing in GitHub notifies, through a Hypertext Transfer Protocol (HTTP) POST request, Jenkins' `ghprb` web service that triggers the CI/CD pipeline associated with the given software component. If successful, a new Docker image is built and published in the Docker Hub repository within the INDIGO organisation. The exit status of the CI/CD pipeline is updated in the GitHub's PR, which will indicate the viability of the change, according to the result of the pipeline.

The CI infrastructure

Figure 6.4 shows the evolution of the Jenkins CI builds carried out, for the static and dynamic tests part of the CI/CD pipelines, over the course of the project. The values for each type of test represent the sum of the total builds of each component from the INDIGO software stack.

The continuation of the SQA activity in subsequent projects, notably in DEEP and eXtreme-DataCloud, enabled the maintenance and operation of this CI infrastructure once reached the project's end of life. Some of the software components not involved in those subsequent projects leveraged this continuance to keep using it for their own developments. In other cases, the developers deployed their own CI systems, taking advantage of the experience gained during

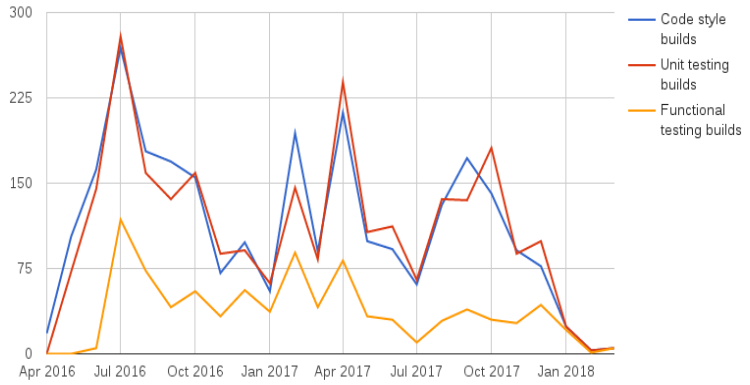

Figure 6.4: Evolution of the total number of builds in Jenkins being triggered automatically as part of the operation of the SQA process in INDIGO. The automated functional testing coverage were not available for all the software stack, thus, as the figure shows, the associated number of builds are fewer. The visible peaks and valleys in the chart correspond to the development efforts prior to the scheduled official releases. Towards the end of the project, the software development activity slowed the pace but not completely ceased.

the project. Hence, either by direct support or education, the INDIGO CI infrastructure contributed to the sustainability of the software that outlived the project.

DevOps for the use cases

The aforementioned CI/CD implementation governed the delivery of the software that made viable the INDIGO services. As such, they are referred to as the core or infrastructure services. On the other hand, the applications from the scientific communities participating in the project relied were integrated in parallel to profit from the new capabilities of such INDIGO core services.

After the successful experiences, especially in terms of reliability and agility, obtained from the above-described DevOps practices in the delivery of the core components, the use cases' applications seemed the next natural step. Consequently, two use case applications, DisVis [140] and PowerFit [141], benefited

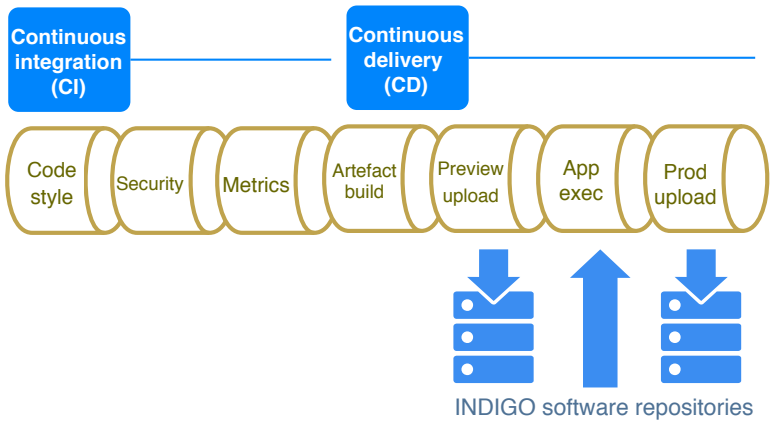

Figure 6.5: DevOps pipeline to distribute Docker images for the DisVis and Powerfit applications. Note that the CI part is a simplified version, where unit and functional tests are missing. This fact results from the lower software testing skills commonly present in computer scientists, when compared with the more engineering-oriented developers of the core components. Nevertheless, the Continuous Delivery (CD) part is an improved version of the pipeline used for the core components, which adds the execution of the application packaged as a Docker image in the previous step.

from those experiences, following a similar approach.

Indeed, the resultant CI/CD pipelines encompassed a simplified version of the CI phase, but instead the CD part of the pipeline were enriched with the validation of the application, once packaged as a *preview* Docker image. The validation process relied on the availability of pre-existing reference outputs, obtained from a previous computation that used the same inputs than the ones now passed to the pipeline. Hence, based on the deterministic nature of both applications, the results of executing the new Docker-based application must be identical to the reference outputs.

Unlike the INDIGO core services, the Docker images were automatically tagged as production versions, since the validation test provided sufficient guarantee.

6.3 Compliance with the requirements from the Software Quality Assurance baseline

The aim of the CI/CD pipelines is to tackle as much requirements from the SQA baseline as possible. Eventually, some of those requirements appeared to be challenging to be addressed by automated means, and in such situations, there was no other way out than to rely on manual assessment, which is impracticable from a change-basis perspective. A recurrent example was the unit or functional testing requirement, whose automation entails a more advanced knowledge.

Code style

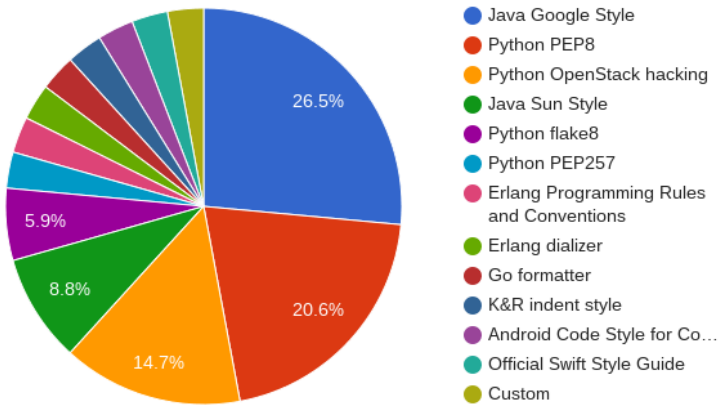

Figure 6.6: Code style standards followed by INDIGO software products.

As it was stated in the SQA baseline's "Requirement X: Adhere your code to a code style standard", both the promotion of the interoperability and readability implies the use of de-facto or community-adopted standards. The heterogeneity and diverse backgrounds of the INDIGO software components hindered the adoption of a unique standard per programming language in use. Figure 6.6 reflects this situation, where different style guidelines were adopted for the same

programming language. Nevertheless, the de-facto standards were the most used, as it was the case for Python's PEP8 or Java's Google Style guidelines.

Unit testing

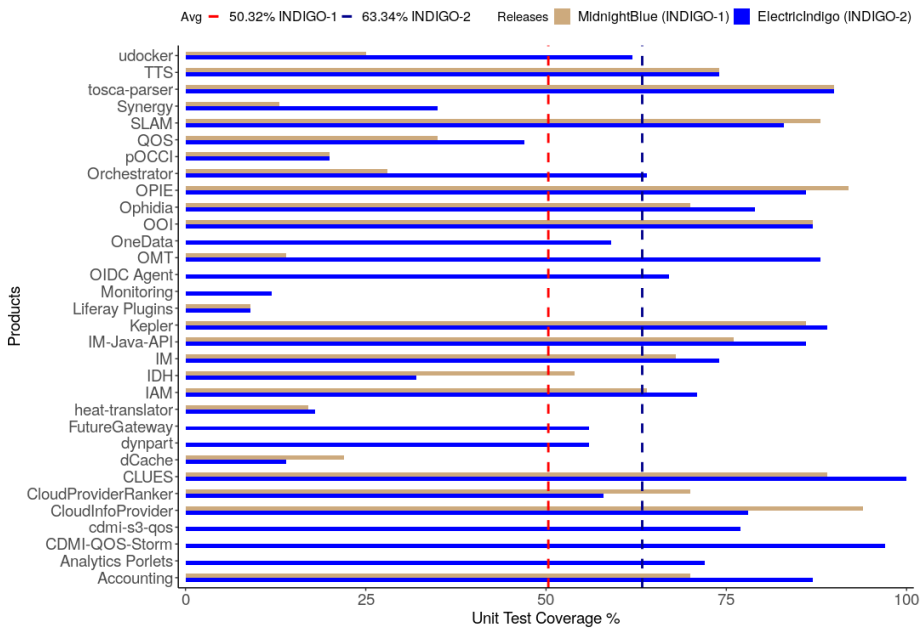

Figure 6.7: Comparison of the unit testing coverage values for the INDIGO software stack over the two major releases, `INDIGO-1` and `INDIGO-2`.

The adoption of unit testing was predominant throughout the lifespan of the INDIGO. In Figure 6.7, unit testing coverage is compared for each software component between the two major releases delivered over the course of the project. As outlined by the figure, there exists an incremental trend at the end of each release that brought the coverage from an average value of 50.32% to 63.34%.

Nevertheless, Figure 6.7 also shows a decrease in the unit testing coverage

values for 18% of products. This decrease was observed to be aligned with high-demanding periods of software development, as showcased in the previous months of the second major release. In software programming, it is commonplace to consider tests as accessory, and thus, in the best-case scenario, they are implemented once the required change in the code has been completed. However, this practice does not comply with the recommended form in "Requirement VI: Test the individual units of the code", where unit tests, as they cover low-level elements of the code, are best considered while the new code is being written. Besides, modern testing methodologies, such as Test-Driven Development (TDD), go beyond this approach to promote writing the tests before the code.

In any case, the application of the SQA process resulted in the overall increase of the unit testing coverage in both the release checkpoints. In the date of the second release, 53% of the software components (17 out of 32) were over the threshold of the aforementioned project's SQA code coverage recommendation (70%), while 75% (24 out of 32) of the product stack exceeded 50% coverage.

Functional and regression testing

Functional testing followed the same continual growth as the unit testing case, but in this case in terms of adoption, not functional coverage. The adoption went from 30% to almost 60% in the time frame between the first and the second major release. As with unit testing, test reports were asked in those cases where automated functional tests were not provided. And similarly, the fulfillment of their fundamental functional requirements was tougher to track, resulting in a non-viable solution for assessing minor changes, especially when analysing the regressions.

The major issues observed that hindered developers from the implementation of automated functional tests were related to the unavailability –at the time– of libraries and frameworks for tackling specific tests –such as testing graphical interfaces–, or in the less representative cases, due to low skilled developers.

Continuous configuration automation

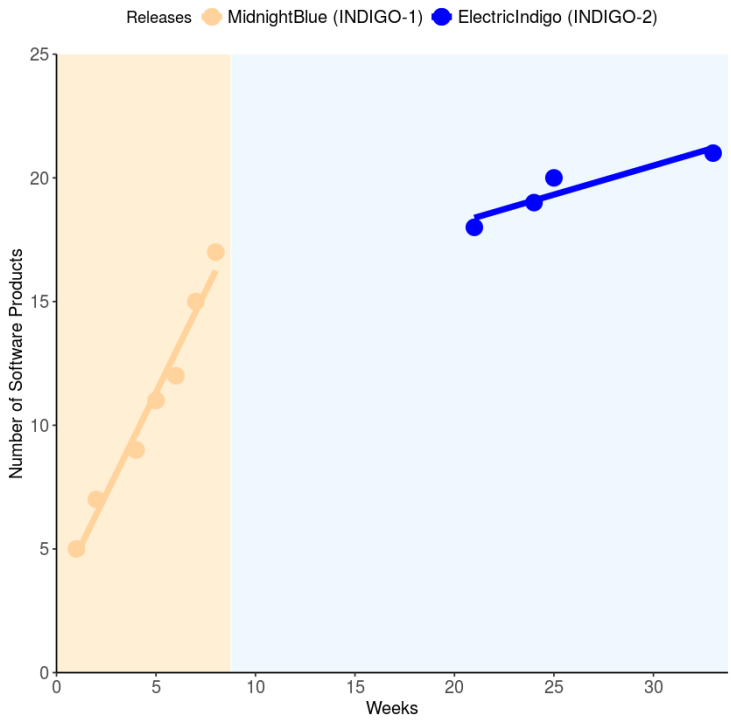

Figure 6.8: Adoption of Continuous Configuration Automation (CCA) tools throughout the project lifetime. The figure shows the trend lines leading to the first (light cream points and line) and second release (dark blue points and line).

In compliance with "Good practice X: Ease the deployment of your software" from the SQA baseline, INDIGO contributed to open–source CCA tools such as Ansible [134] and Puppet [135]. A representative example of such contributions are the 50 Ansible roles developed over the project lifespan, which are hosted in the Ansible Galaxy portal [142].

Figure 6.8 shows the evolution of INDIGO products that adopted a CCA tool for deployment. There is a high rate of adoption in the weeks preceding

the major release dates. The trend is lower during the weeks before the second release because a significant fraction of the products had already adopted it previously to the first release.

Documentation

In addition to the CCA solutions, the documentation is a crucial part of the software adoption. In compliance with "Requirement XIII: Comprehensive documentation", the primary goal in INDIGO was to cover the right set of documents according to the intended audience. This objective is hard to be fully tested using an automated approach, so an oversight by the SQA team was performed at component release time. Furthermore, at the later stage of the preview testbed integration, both the CCA module and the documentation provided by the developers were used to deploy and test the functionality of the candidate component.

What it was fully automated was the generation of the documentation. One indispensable requirement of the SQA baseline is to treat documentation as code, and thus, the INDIGO GitHub organisation contained all the documentation repositories for all the software products. On each update, the documentation, in Markdown format, was automatically generated and rendered in the Gitbook repository [143].

Code review

Following "Good practice VIII: Protect your long-term branches from direct modifications", the source code repositories are protected against direct push operations, so that the code review stage always takes place. Through PRs, code reviewers analyse the content of each change in the source code and their suitability, according to the "Requirement XII: Broadening the perspective with peer reviews of the code". The level of compliance with these guidelines was hard to be gauged. Furthermore, at the time of running INDIGO, GitHub did not provide the feature of setting the minimum amount of peer reviews to

be done within a software repository, so even the ability of enforcing the code reviews was limited.

As already discussed, code reviews are useful when two or three different reviews can be done, and most useful whenever at least one is done by an expert developer not involved in the software project. In INDIGO, a significative part of the software components were supported by small teams, some of them even comprised by a unique member, which hindered the analysis of a different reviewer other than the one implementing the change.

Integration testing

As illustrated in the overview of the INDIGO SQA process, see Figure 6.1, integration testing was undertaken once the CI/CD pipeline delivered the *preview* artefacts in the repositories. As a result, this type of testing was not part of the pipeline, and thus, it was not being carried out automatically.

Also, as stated in Figure 6.2, the integration stage proved to be an effective tool to uncover bugs. The reason behind this fact is that the integration testing comprised the operational part, or in other words, the *Ops* part within DevOps. Prior to the integration checks, the software was required to be deployed in the testbed, which was done by the SQA team using the tools provided by the latter, i.e., the CCA modules and the documentation. This meant that the software was used for the very first time outside the developer's domain, which was reflected by the discovery of bugs or lack of the basic functionalities, not only in the code, but also in the CCA modules and documentation. Figure 6.9 shows the geographically dispersed resource providers that contributed to the testbed and the services maintained at each provider.

The final validation step, also highlighted in Figure 6.1, bolstered the operational readiness of the products. Hence, the software validated in the testbed, was tested by a set of candidate resource providers within the EGI production e-Infrastructure. This process, coined as *stage rollout*, is key to detect and mitigate issues that could only appear in production environments. It will be later on discussed in Chapter 8 "Software validation in the European Grid

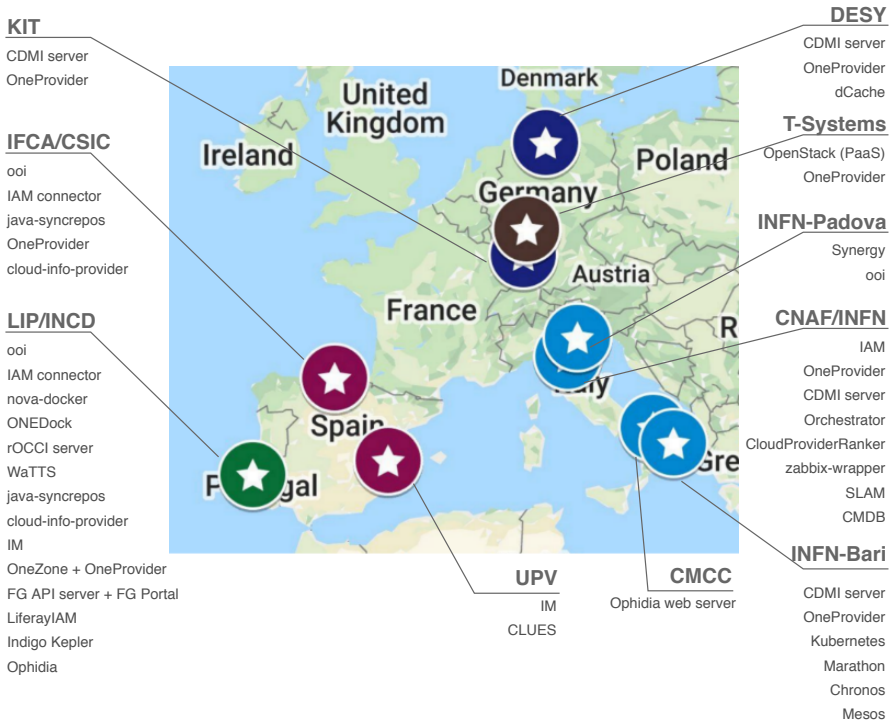

Figure 6.9: Resource centers supporting the pilot preview testbed and corresponding set of deployed INDIGO components or services.

Infrastructure".

6.4 Conclusion

The quality of the INDIGO software products was in constant increase over the course of the project. The setup of CI/CD pipelines and the associated CI infrastructure, to support more than 30 products and 200 GitHub repositories, guaranteed that the changes done at the source code level matched the requirements from the SQA baseline in an automated fashion. The pipelines

delivered the *preview* software artefacts that were subsequently validated by manual means in two successive steps, which included both the local testbed and the EGI e-Infrastructure.

One of the most noticeable metric that demonstrates a reliability improvement is represented by the ratio between the number of software defects uncovered within the INDIGO SQA process, including the integration testing, and once the software had been deployed in EGI. Hence, the number of bugs detected throughout the SQA process surpassed by a factor of 30 the number of bugs reported by those end users of the EGI e-Infrastructure, over a total of almost 200 bugs in the observed time frame. This actually represents a low rate of bugs according to the size of the INDIGO project, and thus, it can be inferred that the automation of herein described stages of the V&V processes had a substantial impact on these numbers.

Nevertheless, the individual products of INDIGO did not undergo a progressive quality improvement throughout their lifespan. Indeed, unit testing coverage dropped for about 18% of the total products halfway through the project. This behavior was aligned with the high-demanding periods of software development, in particular, the previous months before the second major release. This fact demonstrates that the "Requirement VI: Test the individual units of the code" was not actively pursued during certain periods within the project's lifetime.

Automated functional tests followed a continuous growth along the project, reaching the 60% of the total products by the second major release. As it can be concluded from INDIGO outcomes, their provision should be a goal in modern software development as it is key both for the operational impact and towards the final user acceptance. Manual assessment of functional tests was found to be unfeasible for a per-change based SQA strategy.

But automated testing requires from an extensive knowledge in the particular technologies or frameworks that are available for programming languages. During INDIGO, a handful of development teams lacked from the necessary technical skills to profit from these tools, which challenged the application of

the relevant SQA baseline requirements. Just as education is essential towards the increase of quality in the research software, the demonstration of the immediate benefits that these type of tools bring along is crucial to stimulate their adoption.

In this regard, the INDIGO SQA process enlightened developers not familiar with DevOps-driven quality approach to software, proved by the fact that they outlived the INDIGO project. But software engineers in charge of the core products were not the only benefactors of the DevOps culture. The delivery of two applications from the INDIGO use cases were also managed through CI/CD pipelines that, unlike the core products, achieved the provision of the application's artefacts to production repositories through a previous validation of the *preview* artefacts.

The INDIGO SQA process persisted once the project concluded, laying out the grounds of foregoing projects, as it will be discussed in the next chapter. The new software development initiatives contributed to the INDIGO process with a special focus on the inclusion of the use case applications, whose first steps had already being taken.

7

Tailoring software to user needs: the DEEP-HybridDataCloud project

The DEEP Hybrid-DataCloud (DEEP) relied on INDIGO-DataCloud (INDIGO) software products to build a comprehensive cloud-based solution that eased the development, training and exploitation of Machine Learning (ML) applications. The project's goal was to extend the limits of automation to cope with a continuous deployment scenario for such applications that accelerated the readiness of the incoming software releases, in order to be used by researchers via ML inference.

Similarly, the Software Quality Assurance (SQA) process continues its own evolution throughout DEEP project, evolving towards the Pipeline as Code

(PaC) capabilities provided by Jenkins.

7.1 Moving towards a Pipeline as Code environment

The INDIGO Continuous Integration and Delivery (CI/CD) pipelines consisted in server-side definitions set up by the SQA team, composed through a graphical interface provided by Jenkins. As a result, the pipelines' definition were tied to a specific Jenkins instance, and thus, with a very limited capacity of being migrated. This fact proved to be a major drawback when it comes to preserve the SQA outcomes beyond the short-term –commonly no longer than three years– duration of projects like DEEP or INDIGO.

Box 1.1: Benefits of Pipeline as Code

Versioning Probably the most relevant benefit is that the `Jenkinsfile`, or code pipeline, is added to the source code repository, and thus, it is versioned, profiting from all the characteristics provided by a Source Code Manager (SCM) system, as described in "Requirement I: Create versions of the source code".

Active maintenance As the code pipeline is now placed alongside the source code, the developer is quite more involved in the maintenance and enhancement of the pipeline.

Portability Code pipelines enable the seamless operation of the same pipeline in a different Jenkins server, thus enabling the preservation of the software quality advancements over the lifespan of the software. In research software, this means that the operation of the pipelines is no longer dependent on the project duration.

Reuse Just as with the source code, the entire code pipelines, or sections thereof, can be easily reused.

The PaC feature extended the Jenkins pipelines so that they could be defined using a Groovy's domain-specific language (DSL) [144] and placed in a specific file named `Jenkinsfile`. The code pipeline technology, although it may seem negligible at first glance, has been a game-changer not only for the outcomes of DEEP project, but also for the future developments, bringing along a number of advantages enumerated in Box 1.1.

Structure of the code pipelines

The code pipelines split the work into *stages*. Within each stage, a particular action is tackled, which may be required to be completed before the subsequent stages take over. Parallel stages can be defined as well, but the bottom line is that the Jenkins pipelines shall perform a sequential type of work. It is important then to sort those stages according to the overall logic and their individual priority.

```
#!/usr/bin/groovy

@Library(['github.com/indigo-dc/jenkins-pipeline-
    library@release/1.4.0']) _

pipeline {
    agent {
        label 'python3.6'
    }
    stages {
        stage('Style analysis') {
            steps {
                checkout scm
                ToxEnvRun('pep8')
            }
            post {
                always {
                    WarningsReport('Pep8')
                }
```

```
            }
        }
        stage('Security scanner') {
            steps {
                checkout scm
                ToxEnvRun('bandit-report')
            }
            post {
                always {
                    HTMLReport('/tmp/bandit', 'index.html', '
                        Bandit report')
                }
            }
        }
    }
}
```

Listing 7.1: Simplified version of a Jenkins' PaC implementation for the DEEP-as-a-Service (DEEPaaS) software component, developed as part of the DEEP solution. The `jenkins-pipeline-library` is loaded with the `@Library` macro so to import the available methods. Library methods are highlighted in blue, within the `stage` declaration.

```
#!/usr/bin/groovy

/**
 * Run Tox's test environment.
 *
 */
def call(String testenv, String filename=null) {
    opts = ['-e '+testenv]
    if (filename) {
        opts += '-c '+filename
    }
    cmd = ['tox'] + opts
    sh(script: cmd.join(' '))
```

```
}
```

Listing 7.2: The definition of the `ToxEnvRun` method from the `jenkins-pipeline-library`.

Library for the code pipelines

As the code pipelines herein discussed implement the criteria from the SQA baseline, it is commonplace to have very similar definitions for the different software repositories, only varying on programming language specifics. Keeping in line with the *do-not-repeat-yourself* (or *DRY*) principle, Jenkins allows to load custom definitions by means of a shared library.

Leveraging this feature, the main functionalities required to implement the SQA baseline criteria, and other convenient features of CI/CD environments, have been progressively added to an ad-hoc library coined as *jenkins-pipeline-library* [145]. Following the collaborative mandate, of "Requirement II: Use public forges to distribute your work", and based on the fact that most of the SQA baseline criteria are commonplace in other SQA environments, the `jenkins-pipeline-library` has been made publicly available and open to external collaboration since its inception.

Listing 7.1 presents a simplified version of a code pipeline used for a DEEP software component. The library is loaded before the pipeline definition so that the methods can be subsequently used within the stages. Each stage represents a unit of work within the pipeline, usually tackling one requirement from the SQA baseline. One of the library methods used in the pipeline is showcased in Listing 7.2. As it can be noted, the use of the library reduces the size of the pipeline and improves the overall readability.

7.2 Stage composition of Continuous Integration and Delivery code pipelines

As in the case of INDIGO, DEEP is committed to fulfill the SQA baseline criteria for any software development effort within the core components of the DEEP solution. Building on the work started in INDIGO, all the DEEP use cases, i.e. the DEEP-ML applications, implemented, in a first iteration, a similar approach as the one described in the previous chapter for the two INDIGO use cases. Hence, the Continuous Integration (CI) part was more relaxed than the core components, yet still more advanced than the previous INDIGO use cases.

Agile integration of the continuous releases

The code pipelines for the core components were similar, in terms of requirements, as their counterparts in INDIGO. The most noticeable improvement is done at the integration-time takeover. At INDIGO, there existed a misalignment among the automated and manual phases –i.e., when the preview artefacts are delivered to be validated in the testbed–, manifested by a communication gap that prevented from a more agile validation of the incoming artefacts.

To mitigate this gap, as shown in Figure 7.1, the DEEP approach is to extend the CI/CD pipeline with a new *notification* stage. This stage leverages the DEEP project management tool in order to automatically create an issue for every new release. The issue includes all the relevant information including the release tag and the artefacts location. The preview testbed manager is immediately notified when the issue is created, so the communication breach is cut down. Additionally, the pipeline also added as issue watchers the representatives of the user communities present in the project, so as to be aware and try by themselves the new release of the component once integrated in the testbed.

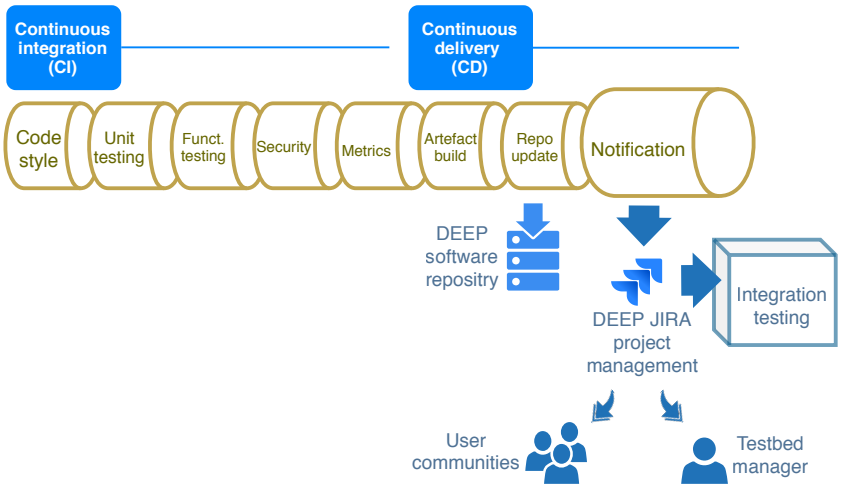

Figure 7.1: Pipeline for the core components in DEEP, where the notification stage is highlighted. Once the preview artefacts are uploaded to the repositories, the pipeline creates a new issue through the Application Programming Interface (API) exposed by the JIRA project management tool. Stakeholders are then immediately notified of every successful software delivery done by the CI/CD pipelines.

Interplay of code pipelines for the delivery of DEEP-ML applications

DEEP solution was designed to deal only with Docker containers, so accordingly, applications are only distributed as Docker images. The instructions to build these Docker images –compiled in a `Dockerfile`– are maintained in an individual code repository, thus decoupling the CI and CD pipelines, and simultaneously, the application code from the delivery settings. In particular, the CD pipeline does not only implements the Docker image management, but also the identification of the given DEEP-ML application in the project's marketplace or DEEP Open Catalogue (DEEP-OC). As such, it will be referred throughout the document as DEEP Open Catalogue application (DEEP-OC-app).

The rationale behind this layout lies on the need of rebuilding the DEEP-ML applications not only as a result of changes in the source code, but most importantly, in the event of new releases of the DEEPaaS component. The DEEP-

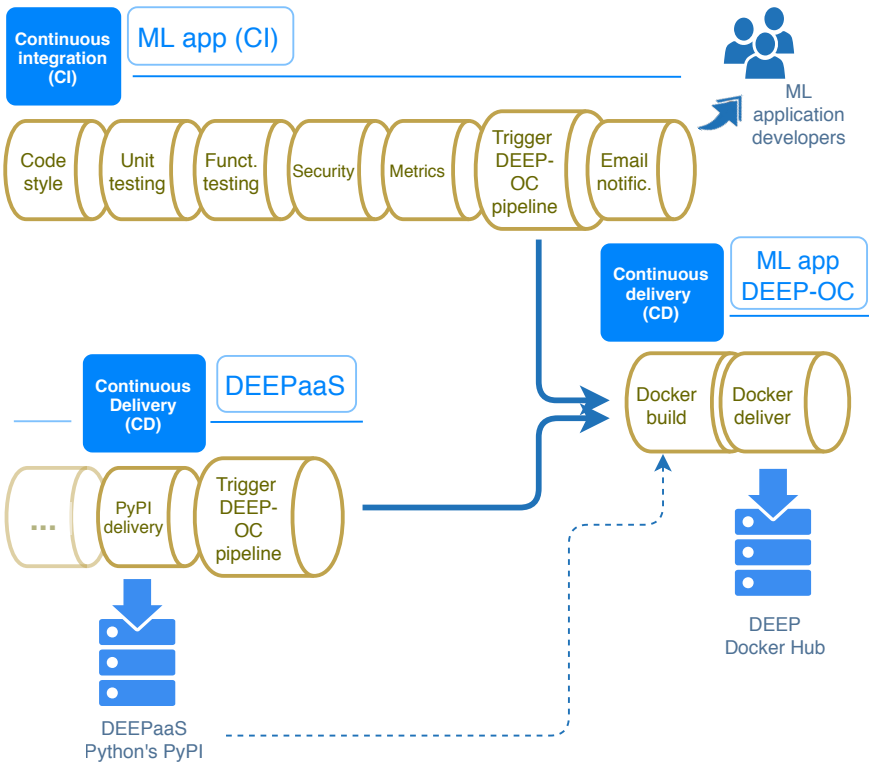

Figure 7.2: Interplay between the three code pipelines in order to deliver DEEP-ML applications. Decoupling the Continuous Delivery (CD) phase, represented in the figure as *ML app DEEP-OC*, facilitates its activation by the DEEPaaS pipeline whenever new releases of this component are delivered. When this happens, the new version of DEEPaaS is automatically pushed to Python's PyPI repositories, which is subsequently used to rebuild the DEEP-ML Docker image.

aaS software provides the RESTful API through which the main functionalities are made available, and accordingly, when the DEEPaaS API changes, all the Docker images of the DEEP-ML applications shall be updated. Figure 7.2 illustrates the pipeline breakdown, together with the automated interactions among them.

7.3 Extended automation beyond Continuous Integration and Delivery environments

The adoption of the PaC, or code pipelines, described in the previous section, brings along the demonstrated benefits enumerated in Box 1.1. In this section, the most significant new breakthroughs, enabled by the use of this technology, are presented. They harness automation to advance towards more sophisticated DevOps models, i.e the continuous deployment of DEEP-ML applications, seamlessly accessible through the DEEP-OC.

7.3.1 Automated generation of Open Catalogue's content

As in the case of the European Open Science Cloud (EOSC) portal, from Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud", marketplaces are the access points for customers to be informed of the new developments and make use of the services. The DEEP-OC presents an updated view of the available DEEP-ML applications, with related information, and the guidelines to run, train and use them.

The website is rendered using a static site generator that significantly simplifies the management of the content. The entire website is version-controlled and managed with a Jenkins code pipeline. Accordingly, upon modification, the HyperText Markup Language (HTML) content is automatically rebuilt by the site generator, and right after, the new changes are displayed in the website. To avoid dealing with HTML code, the applications' description are maintained in individual Markdown files, showcasing their background, motivation and usage.

Metadata for DEEP-ML applications

The aforementioned process was simple enough for the reduced set of pilot applications gathered around the project's proposal, but it required application maintainers to create a new Pull Request (PR) in the catalogue's repository for every single modification in the description of the application. Hence, in light of

forthcoming requests from external research communities to join the DEEP-OC, the need of ***separating the descriptions of the DEEP-ML applications from the website*** itself became clear, thus giving the developers full control and accountability of such descriptions.

To this end, the structure of the DEEP-ML application's description needs to be defined through a *schema*. A schema enables the representation of structured data, by means of formatting rules that define fields are expected, and how the corresponding values can be provided. The developed DEEP schema [146] relies on JSON schema implementation [147].

Once having the schema, the JSON metadata was composed with the accurate description of each application. The metadata was added to each DEEP-OC-app code repository. Hence, the application developer is able to manage any prospective modification through version control, and the results will be readily displayed in the DEEP catalogue as described below.

Open Catalogue content

Three main actions were needed in order to adapt the DEEP-OC to the new demands. First, the metadata must be validated before being subsequently processed. Second, this metadata is required to be converted into Markdown format that, as explained in the introduction of this section, is the human-readable format provided by the web framework. As a last step, a registry shall be maintained, in order to manage the right set of applications that shall be accessible through the DEEP-OC. Evidently, any modification to this registry must be reviewed and approved by the DEEP-OC manager as a precautionary step to avoid undesired applications being accessible through the catalogue. Box 3.1 summarizes the solutions adopted for each action.

The new scenario required an increased amount of interactions among the code pipelines. Now the website content is spread over the DEEP-OC-app repositories, which required an additional stage in order to trigger a website rebuilding each time their own metadata changes. Moreover, the DEEP catalogue's pipeline manages the registry of production applications so it needs to

implement the logic to react to changes in this file. Figure 7.3 schematizes how the different workflows are triggered by the Jenkins pipelines involved in this scenario, i.e. both the DEEP-OC's and the DEEP-OC-app repositories.

Box 3.1: Implementation of the catalogue requirements

Metadata validation . The DEEP validator [146] uses Python's `jsonschema` module [148] to validate to the JSON's Draft 7 specification [149] used by the DEEP schema.

Metadata to Markdown . JSON-formatted metadata is converted to Markdown format by means of a templating script added to the DEEP-OC's code repository [150].

DEEP application's registry . The registry file is YAML-based file, coined as `MODULES.yml`, that contains the list of DEEP-OC-app repositories that ought to be accessible through the DEEP-OC. Accordingly, the registry is also managed through the catalogue's repository [150]. This repository is protected from direct modifications, following the "Good practice VIII: Protect your long-term branches from direct modifications", and thus, a change in the registry file must go through code review.

7.3.2 Continuous Deployment for machine learning inference

As briefly introduced in the previous section, the DEEPaaS component provides the DEEP-ML applications with a ready-to-use RESTful API that enables training or inference –where the prediction of a trained model comes into play– operations through Hypertext Transfer Protocol (HTTP) calls. The DEEPaaS component is an integral part of the overall DEEP solution.

The DEEP-ML applications deploy the DEEPaaS component as part of the Docker image building process. If the *dockerized* application is available as a long-running service, it would need to be re-deployed every time the image is

rebuilt, according to the pipeline workflows described in Figure 7.2. This task is efficiently tackled, through rolling updates or restarts, by nowadays' container orchestration frameworks.

The view of having the last stable versions of all the DEEP-ML applications, available through the DEEP-OC, continuously running and automatically updated every time a new version of the application was released, appeared as an apparent but challenging step forward. The process would involve to extend the current workflow defined in the code pipelines to invoke the rolling update operation for the list of production DEEP-ML applications accessible for inference through the catalogue.

Rolling update through Function as a Service (FaaS)

To this end, a FaaS was set up so that the DEEP-ML applications could be accessed through a single DEEPaaS endpoint [151]. The FaaS exposes an *update* function that will take care of performing the rolling update in the orchestration framework.

Box 3.2: Criteria for adding/removing ML applications

1. The DEEP-ML application must be part of the DEEP-OC. Accordingly, if an application accessible through the FaaS is removed from the catalogue, it will be removed as well from the DEEPaaS endpoint.
2. The DEEP-ML application must be compatible with the DEEPaaS API version deployed in the DEEPaaS endpoint.
3. The DEEP-ML application must be trained.

The DEEP-OC code pipeline is the one in charge of triggering the FaaS update function. But before, the pipeline has to handle yet another file, this time the FaaS configuration file, containing the list of DEEP-ML applications available for model inference. The apps available for inference through the DEEPaaS endpoint might not necessarily be the same as the complete list of applications

accessible through the DEEP-OC. The application readiness is governed by the criteria in Box 3.2.

Figure 7.3 depicts the new stages of the DEEP-OC pipeline, including the two main improvements being described in the above sections, i.e. both the management of the catalogue registry and the FaaS configuration file for the DEEPaaS endpoint. Furthermore, the figure illustrates the interplay with the DEEP-OC-app repository pipeline, as one of the possible ways to trigger the DEEP-OC pipeline.

7.4 Conclusion

The DEEP project represents a continuation of the work initiated within INDIGO project with regards to the establishment of the quality habits in software production, both through the maintenance and enhancement of the SQA baseline and its practical application. In this regard, the adoption of the Jenkins' PaC technology enabled the decoupling of the pipeline definitions from server-side, moving them alongside the repository of code. As the pipelines are now owned by software developers, these latter become ***more aware and committed to the realisation of SQA practices***. Furthermore, as pipelines are no longer dependant of the given Jenkins instance where they were defined, the software is now ***safe from the discontinued operation of short-lived development projects***. The quality assessment is continued beyond the project the software was first implemented, contributing to its long-term sustainability.

The pipeline definition has been simplified, as well as its readiness improved, through the implementation of the `jenkins-pipeline-library`. The library aims primarily at fulfilling the common operations required by the practical implementation of the requirements coming from the SQA baseline, but eventually extended functionality was added. The library is publicly available in order to ***contribute to applied SQA realisation***.

But the major breakthrough, in terms of software quality realisation, refers to the advancement in extending the previously reached limits of software au-

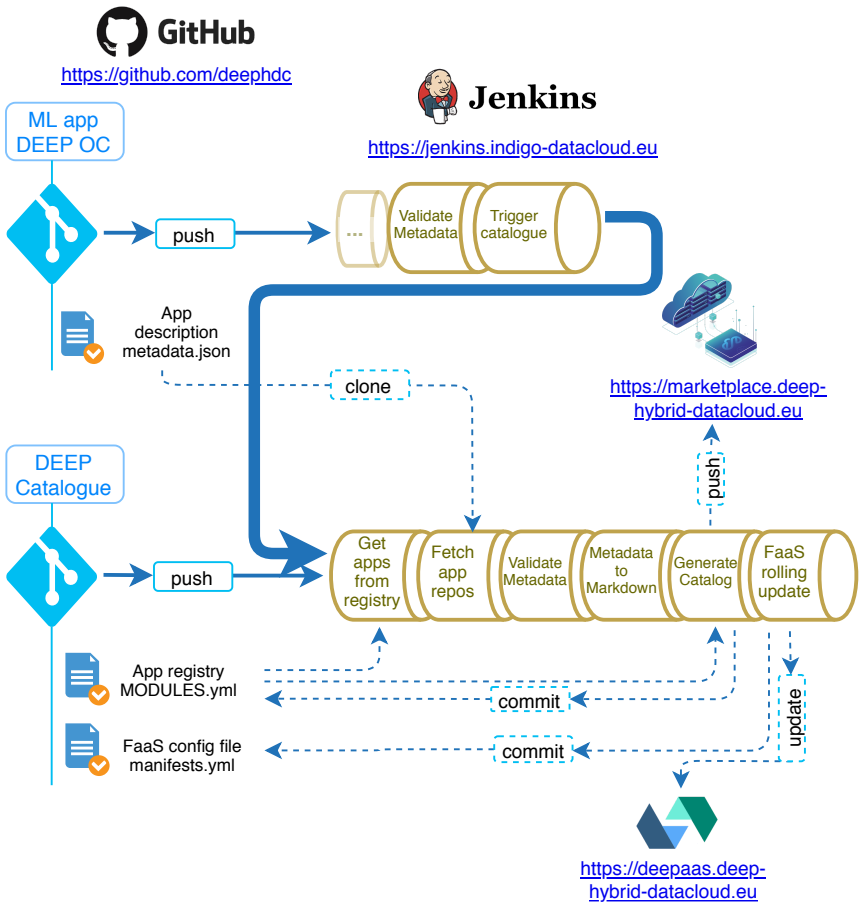

Figure 7.3: A comprehensive illustration of the pipelines (and stages) that manage the generation of both the DEEP-OC content and the applications available through the DEEPaaS endpoint. This pipeline is not only triggered by modifications in the DEEP-OC repository, but also as a result of changes in the DEEP-ML applications' metadata.

tomation and DevOps culture, putting the focus on improving the user experience through the continuous deployment of the services being offered to researchers. This fact was motivated by the improved flexibility of the code

pipelines, which enable the realisation of more complex tasks. Based on the described milestones, the pipelines drive the life-cycle of the DEEP-ML applications, covering the i) development –carrying out the quality Verification and Validation (V&V) processes–, ii) the delivery as Docker images, iii) the publication in the DEEP-OC, and, as the final element in the equation, iv) the application deployment. The first two phases within the DEEP application life-cycle built on INDIGO outcomes, but the last two capture the very essence of what was contributed by the DEEP project.

On the one hand, the ***DEEP-OC is fully managed with a Jenkins code pipeline***. The applications accessible through the catalogue are described by their owners through a metadata file –compliant with a custom DEEP JSON schema– located in the application code repository. Application developers are then responsible for the appearance of the application within the catalogue, but nevertheless, the ultimate decision about their availability is managed through the catalogue's registry. Any change to this registry is reviewed by the DEEP-OC's manager so that, on approval, the catalogue's pipeline is triggered in order to re-generate the website. Additionally, this pipeline can be triggered by changes on the applications' metadata.

On the other hand, the ***last stable versions of the DEEP-ML applications are readily available for model inference*** through the DEEPaaS endpoint. By leveraging a FaaS solution, the pipelines maintain the list of inference-ready applications that the FaaS framework uses for the deployment. Consequently, in the event of changes in this file –induced by previous changes on the catalogue's registry or in the application's metadata– the catalogue pipeline triggers the FaaS in order to make it aware of the new changes. FaaS then takes care of deploying the right versions, through rolling updates. The DEEPaaS endpoint for each application is accessible through the DEEP-OC.

As a consequence of the achievement of these two milestones, ***freshly generated research value can be readily exploited by the scientific communities*** –either internal or external to the project– through an extended continuous delivery and deployment approach. Therefore, the research value is no longer

delivered at the end of the research project lifespan –or in a reduced and fixed basis–, but readily accessible across its duration. Additionally, the uptake of new research is also empowered by the described process, as the ***integration of new DEEP-ML applications in the loop is fully automated*** upon approval of the project management.

8

Software validation in the European Grid Infrastructure

Part of this chapter has been published as:

Pablo Orviz Fernández, Joao Pina, Álvaro López García, Isabel Campos Plasencia, Mario David, and Jorge Gomes. "umd-verification: Automation of Software Validation for the EGI Federated Infrastructure". In: **Journal of Grid Computing** ***16.4 (2018). Quartile: Q1, JIF Percentile: 76.452, pages 683–696. DOI:*** *10.1007/s10723-018-9454-2*

From an e-Infrastructure perspective, the operational stability of the services being offered to researchers is the most outstanding objective. Therefore, in the DevOps culture, the e-Infrastructures, such as the European Grid Infras-

tructure (EGI), become the *Ops* while the previous DEEP Hybrid-DataCloud (DEEP) and INDIGO-DataCloud (INDIGO) projects are the *Devs*. In anticipation to these circumstances, the Software Development Life Cycle (SDLC) adopted throughout DEEP and INDIGO project was purposely oriented to meet the DevOps principles, and thus, really emphasizing the processes that fostered the accurate operation of the services once deployed on these e-Infrastructures.

However, although the described Software Quality Assurance (SQA) processes are based on state-of-the-art software engineering methodologies, transparent and trustworthy, the e-Infrastructures cannot completely rely on those and need to put in place their own, more lightweight, quality procedures to secure the operation of the incoming software. In this chapter, the focus is set on the software validation within the EGI e-Infrastructure, as a previous step before being deployed as services within the federated resource providers. As the driving force of this thesis, automation plays a central role in the practical implementation of the validation strategies put in place under the EGI Software Provisioning Process (EGI SWPP).

8.1 Software distribution in the European Grid Infrastructure: the Software Provisioning Process

As introduced in Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud", EGI started off from the previous work in Enabling Grids for E-sciencE (EGEE) project, whose grid-based e-Infrastructure was relying at that time on the gLite middleware distribution [153]. *Middleware* was the term used to describe the set of software components that enable the federation of the distributed and heterogeneous resources of a grid at the computing, data management and security levels.

Unified Middleware Distribution

One of the foundations of the EGI paradigm shift was the establishment of a joint grid middleware consortia that eventually gave rise to the European Unified Middleware Distribution (UMD) [154]. UMD supplies grid middleware developed by external Technology Providers (TPs) in order to enhance the operation of the e-Infrastructure. During the first years of operation, the European Middleware Initiative (EMI) [155] –which included the principal European grid middleware providers, such as gLite, UNICORE and ARC– was the central TP of UMD. Once the EMI project ended, several technologies were no longer supported, while others maintained individual support. UMD was first released in July 2011, by means of `UMD-1.0.0`, delivering around 30 grid middleware products.

Cloud Middleware Distribution

Since its inception, EGI has adapted its service offerings to the new computing models beyond the traditional grid technology. In 2014, the EGI Federated Cloud [156] was officially launched, with an architecture based on Infrastructure as a Service (IaaS) cloud service model. In order for cloud providers to join the federation, additional services are needed beyond the ones of the underlying IaaS framework [1].

These services encompass both computing and data management, as well as federation-enabling services, such as authorization and authentication, monitoring, accounting or information discovery. The Cloud Middleware Distribution (CMD) [157] was launched on December 2016 –through the Openstack's (sub-)distribution `CMD-OS-1.0.0` release– containing the software components required for the federation of cloud providers within the EGI Federated Cloud.

[1] EGI FedCloud started off with support both for OpenStack and OpenNebula cloud management frameworks, but at the time of writing, OpenNebula support has been almost decommissioned.

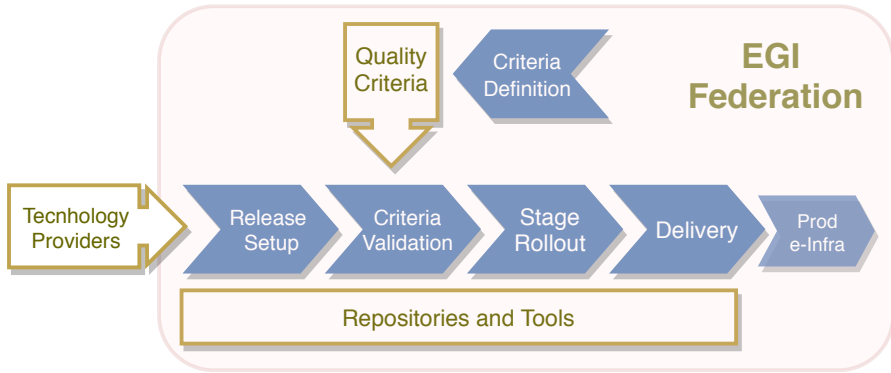

Figure 8.1: The EGI SWPP.

The EGI Software Provisioning Process

UMD and CMD release software that is potentially interesting for the EGI interests, thus motivated by both user and operational requirements. The SQA processes that guided the SDLC of the services are not under the control nor monitored by EGI, so accordingly, there is no guarantee that the software is reliable enough for a production infrastructure. The EGI SWPP [158], schematized in Figure 8.1, validates the incoming software in order to lessen the odds of disruption once it is in production.

The EGI SWPP encompasses the i) *validation of the conformance criteria*, the ii) *staged rollout* phase, which takes over the deployment and user-level testing on production facilities, and, finally, the iii) *release to production*, resulting in the software release preparation and delivery. During the validation of the conformance criteria phase, every piece of software is deployed and tested to detect any malfunction or deviation from the design specification. The procedure of validation is governed by the Quality Criteria (QC) definition, which enforces the quality requirements that any software released under UMD and CMD distributions must comply.

The validation phase appears as the most time-consuming task within the

EGI SWPP since a major effort is spent on dealing with the deployment peculiarities of each software component, as well as in ensuring a minimal testing coverage of the expected functionalities. The next two sections present the two subsequent iterations towards the modernisation of the EGI SWPP, and in particular, within the validation of the conformance criteria phase.

8.2 Phase 1 of the Software Provisioning Process modernization: boosting the software validation

8.2.1 Statement of the problem

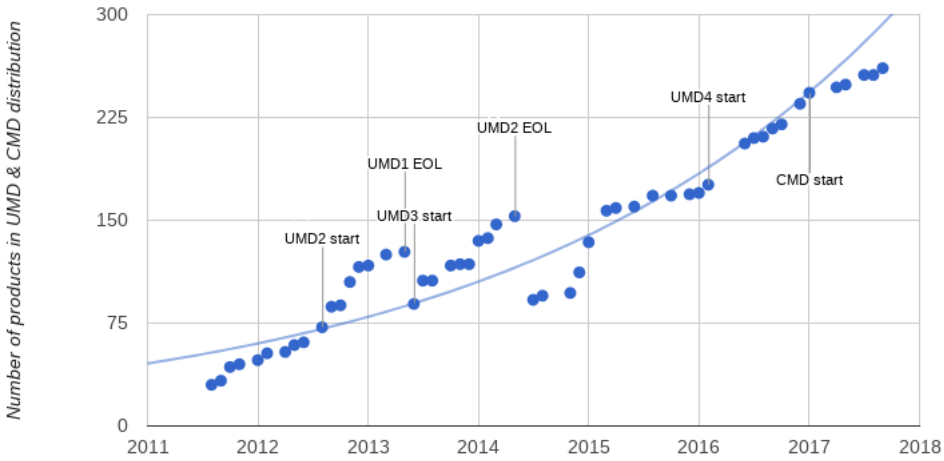

Figure 8.2: Trend graph showing the number of products supported in the EGI production repositories (UMD and CMD). The incremental trend is interrupted by the end-of-life (EOL) cycles, which are rapidly recovered as a result of the parallel start of the subsequent major release version. At this point in time, the incoming UMD major release progressively adopts, following the validation process, the products previously existing (source: repository.egi.eu).

An analysis of the evolution of EGI software product catalogue, outlined in Figure 8.2, shows a ***growing trend in the number of products being supported*** since the first major release of the UMD distribution, `UMD1`. The underlying reasons behind this growth are mainly the evolving technology demands coming from the scientific communities leveraging the EGI e-Infrastructure. More recently, as cloud computing became more popular, these user requirements resulted in the advent of the CMD distribution, which increased considerably the number of products supported, and thus, the equivalent amount of validations [2]

Addressing the growing needs with the former validation process resulted in delays within the EGI SWPP chain, leading to extreme situations where a product release was disregarded and superseded by a subsequent release while queued at this stage. According to [158], the validation of the conformance criteria phase was driven by a team of 15-20 testers, each taking over the product validation process based on their expertise. The process was fully manual, with a typical estimated time completion of 1 or 2 working days for each software validation. Thereby, the traditional approach of the EGI Quality Criteria (EGI QC) validation is only sustainable as long as the `manpower:number of products` ratio remains balanced, which is likely to become unsustainable over time, based on the trend discussed above.

8.2.2 Automation of the Quality Criteria requirements

The adoption of automation seems to be an obvious choice to address the previously identified delays within the validation phase, based on the arguments enumerated in Box 2.1. However, the first step is to assess the feasibility of the automated validation of the requirements that indicate the viability of the product to be deployed in the EGI e-Infrastructure.

[2] It is important to underline that Figure 8.2 only shows the total products, not the actual validations being performed. These value increases according to the number of operating systems being supported by each product. Appendix C.1 details the evolution of the operating systems being supported within UMD and CMD.

Box 2.1: Fundamental arguments to automate the EGI SWPP

Manpower Taking into consideration the above-mentioned fact of requiring 2 working days for each `product:OS` validation, in the likely event of having 20 queued products supported in 2 different Operating Systems (OSs), approximately 80 working days would be needed to complete their validation. Distributing the load among the 15-20 testers, the process would take roughly a full-time week of work for all the members in the validation team.

Expert dependence Whilst the good progress of a manual software validation is driven by seasoned teams, the programmatic implementation of a product validation would only require from expert knowledge the first time it is set up. Once in place, the process could be taken over by non-expert testers since most of the complexity is hidden.

Human factor In the context of mechanical or repetitive processes, the likelihood of human error is substantially higher than when the same process is performed in an automated environment. Whilst automated processes are predictable, humans are not able to work with the same level of consistency.

Time efficiency Automation streamlines the time required to complete a task. Time efficiency is usually associated with automation since it allows to meet strict deadlines or even increase the number of tests that could be performed in the same time slot, resulting in higher test coverages.

The EGI QC document [94] drives the validation of software products within the EGI SWPP workflow. In a much lightweight fashion than the criteria presented for the SQA baseline in Chapter 5 “Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software”, the EGI QC criteria defines the quality requirements that a given product has to fulfill in order to be considered ready for the subsequent staged rollout phase. The readiness of the EGI QC criteria to be automated is summarized in Table 8.1, which lists

those quality requirements, their associated criticality and the possibilities of automation.

8.2.3 The `umd-verification` tool

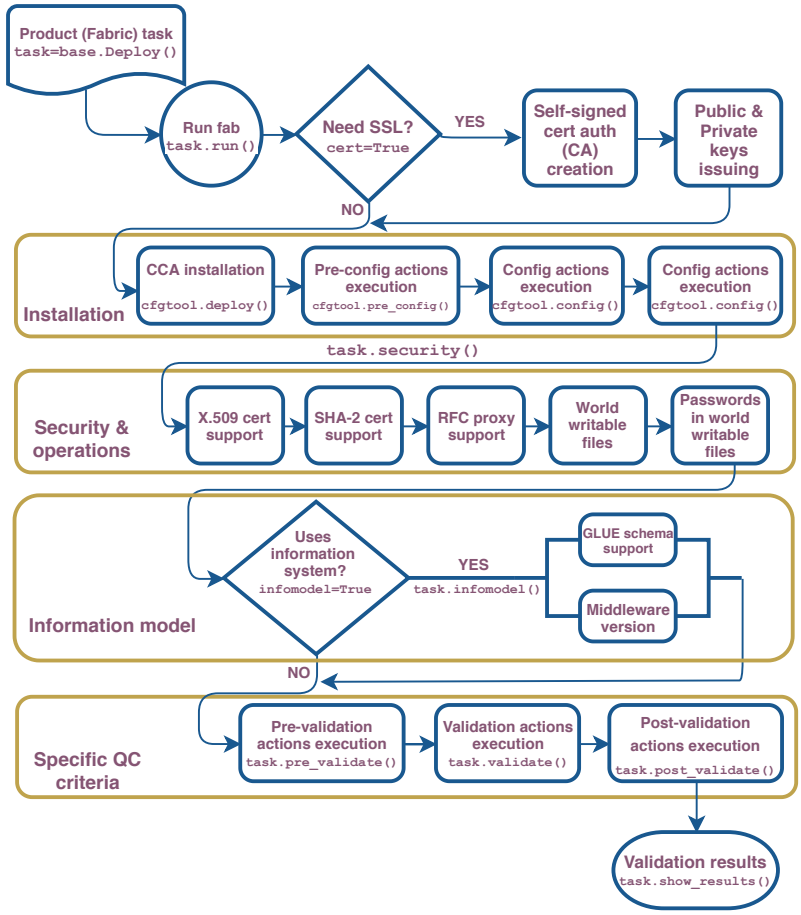

Figure 8.3: Product validation workflow in `umd-verification`.

In order to automatize the software validation process within EGI, the es-

ID	Check	Critical	Automated
	Documentation		
QC_DOC_2	User documentation	✓	✗
QC_DOC_3	API documentation	✗	✗
QC_DOC_4	Admin documentation	✓	✗
QC_DOC_5	Software license	✓	✓
	Installation		
QC_DIST_1	Binary distribution (RPM, DEB)	✓	✓
QC_UPGRADE_1	Upgrade previous working version	✗	✓
	Security		
QC_SEC_1	X.509 certificate support	✓	✓
QC_SEC_2	SHA-2 certificate support	✓	✓
QC_SEC_3	RFC proxy support	✗	✓
QC_SEC_4	ARGUS auth integration	✗	✓
QC_SEC_5	World writable files	✓	✓
QC_SEC_6	Passwords in world readable files	✓	✓
	Information Model		
QC_INFO_1	GLUE schema 1.3 support	✗	✓
QC_INFO_2	GLUE schema 2.0 support	✓	✓
QC_INFO_3	Middleware version	✗	✓
	Operations		
QC_MON_1	Service probes	✗	✓
QC_ACC_1	Accounting records	✓	✓
	Support		
QC_SUPPORT_1	Bug tracking system	✓	✓
	Specific QC		
QC_FUNC_1	Basic functionality test	✓	✓
QC_FUNC_2	New feature/bug fixes test	✗	✓

Table 8.1: EGI QC (v7) requirements. In terms of automation capability, the only requirements that need human interaction are the ones related to the analysis of the documentation (`QC_DOC_x`): one could address programmatically the existence of the required documentation, but not the suitability of its content.

sential component would be a general purpose tool that programatically iterates over the EGI QC requirements, executing the appropriate tasks for analysing their compliance, allowing the process to stop according to the level of criticality. This tool shall rely on Continuous Configuration Automation (CCA) tools to automatically deploy the candidate software products, and eventually ran the tests that ensure the minimal functional feasibility.

The `umd-verification` tool [159] is the solution proposed for the automated, sequential validation of the requirements defined in the EGI QC document. The tool is written using the Python's Fabric library [160] that provides a convenient way to issue high-level system calls. Besides, *fabric-ed* applications are organized in tasks that can be mapped to the validation of the individual EGI software products, and subsequently called via the built-in command-line `fab` tool.

Listing 8.1 shows an example of a Fabric's CMD software product task. Such a task inherits from the `base.Deploy` class that implements all the different execution blocks that encompass the validation of a product according to the EGI QC requirements.

```
from umd import base
from umd.base.configure.ansible import AnsibleConfig

cloud_info_provider = base.Deploy(
    name = "cloud-info-provider",
    doc = "cloud-info-provider␣deployment␣using␣Ansible.",
    cfgtool = AnsibleConfig(
        role =  "https://github.com/egi-qc/ansible-role-cloud-
            info-provider",
        checkout = "umd",
        tags = ["untagged", "cmd"]),
```

```
    qc_specific_id = "cloud-info-provider")
```

Listing 8.1: A task definition for the validation of the CMD's cloud-info-provider product. The task relies on an Ansible role for the deployment and subsequent execution of the functional tests. These tests are defined in a separate configuration file, identified by the label pointed by the `qc_specific_id` attribute.

Behind the scenes

Figure 8.3 shows the tool's workflow. As introduced above, the `base.Deploy` class implements the four major execution blocks highlighted in the figure, i.e. the i) installation and configuration, ii) security and operations, iii) information model, and iv) specific QC. Note that, as already discussed, documentation requirements need of human revision and thus are not being validated by the `umd-verification` tool.

The first block, *Installation*, addresses the deployment of the product using a CCA module. The `base.Deploy._deploy()` method first installs the CCA tool and sets the required environment, such as managing the module –and associated dependencies– installation and generating the input parameter files needed for the module execution. The deployment is then triggered through the `base.Deploy.config()` method, with optional pre- and post-steps that could have previously defined at instantiation time.

The *Security and Operations* block is comprised of a set of basic security assessments. This phase is specially significant for the secured products since it checks the compliance with X.509 cryptographic standard and SHA-2 signatures [161].

Workload orchestration within EGI e-Infrastructure relies on the resource information published by the providers. The *Information Model* block ensures the presence of published resource information, in GLUE format [162] As not all the supported products in UMD and CMD publish GLUE data, the class attribute `has_infomodel` signals when this requirement should be checked.

The last block, *Specific Quality Criteria*, covers the functional and/or integration testing of the product. Here, basic operation and new features and/or bugfixes included in the release are tested. The class attribute `qc_specific_id` maps to the set of tests, in the form of scripts, that must be executed. In the subsequent product validations, these checks eliminate the regression risk as they are re-executed to ensure that the previous working functionalities are kept.

8.2.4 Evidence of the `umd-verification` adoption

Continuous validation of EGI software

`umd-verification` is suitable for being used as part of a Jenkins pipeline. The Jenkins Continuous Integration (CI) system fires up the virtual resource, sets up the tool, triggers the validation of the candidate software with the appropriate runtime parameters and, finally, tears down the provisioned resource. Automation, through the usage of a CI service to take over the validation of products, notably ***hides the inner complexity of the validation process*** –i.e. resource provisioning, `umd-verification` deployment and execution–, ***allowing a non-expert usage***.

Time efficiency for the validation process

The paramount benefit of automating the validation process via the `umd-verification` application is the time efficiency. Combined with the automated resource provisioning, provided by the CI implementation previously described, this efficiency raises even higher. As it was mentioned in the statement of the problem in Section 8.2.1, back in the days of the manual validation process, a common completion time was estimated to be 1 or 2 days. With the new approach the validation process takes a few minutes, although this duration is tightly related to the deployment requirements of each software component, as some products need additional services for the testing phase.

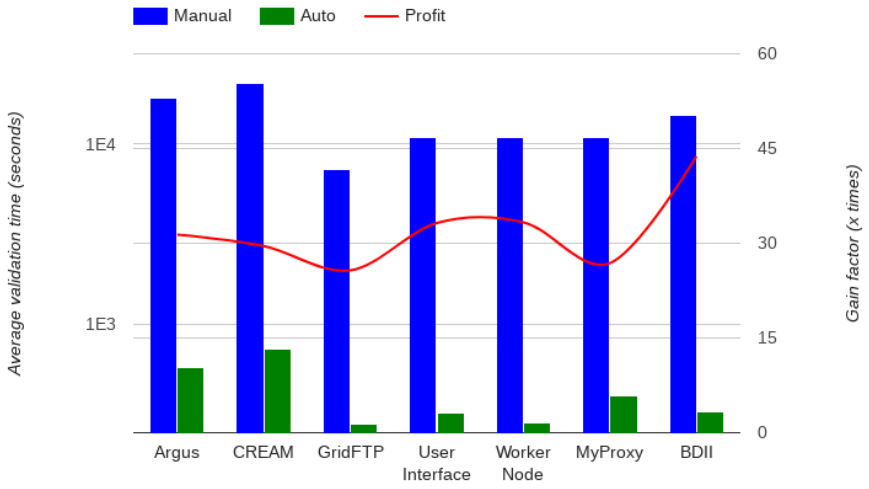

Figure 8.4: Automated vs Manual validation process times. Time values on the vertical axis use a logarithmic scale to better showcase the important differences of time completion for both types of validation processes.

The data displayed in Figure 8.4 compares the validation time of both approaches for a set of UMD products, showcasing the profit percentage obtained with the automated process. The results show an ***average factor of 32 in the time efficiency of the validation process*** with the adoption of the automation process described throughout this paper.

CCA knowledge base

One of the requirements imposed when supporting a new product validation in the `umd-verification` application is the usage of an CCA solution for its deployment. Since the adoption of automation, the EGI validation team maintains a public repository [165] with a collection of Ansible and Puppet modules, summarized in Figure 8.5, resultant from the validation process.

These ***CCA modules can be then reused by the site operators within the EGI e-Infrastructure*** once the software is released through UMD and

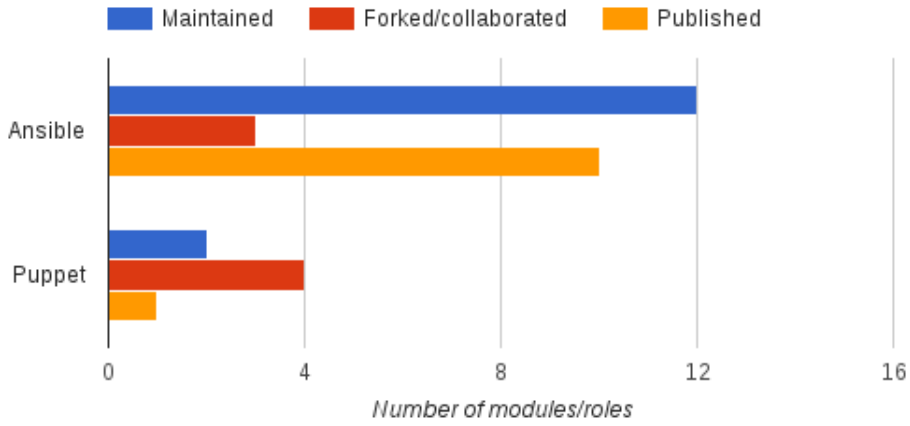

Figure 8.5: CCA modules being maintained, forked and published in the official repositories by the EGI validation team. *Maintained* refer to CCA modules created and supported by the EGI validation team, *forked* are the modules modified and contributed to upstream, and *published* considers the modules contributed to the official Ansible [163] and Puppet [164] community repositories.

CMD repositories. This contrasts with the previous manual procedure, where the work done at the validation stage was not as profitable, having the only reference of a sometimes non-structured documentation being included in the validation report.

8.3 Phase 2 of the Software Provisioning Process modernization: DevOps adoption

The EGI SWPP is set out as an independent process, removed from the previous SDLC step, based on the impossibility of governing, or even monitoring, every SDLC implementation for the whole set of products that are distributed through the EGI repositories. As such, the TPs announce each new release of their own products, and EGI takes care of the remaining activities through the EGI SWPP operation.

Consequently, the EGI SWPP prevents, to a certain extent, the implementation of a real DevOps process, as the TP –the *Dev* in this case– is disconnected from the operational part. Once tackled the automation of validation process in previous 8.2 section, the most costly step in the EGI SWPP, the second phase of the modernisation of the EGI SWPP entails the achievement of continuous validation process, having the TP as the driver.

8.3.1 From release preparation to stage rollout

The EGI SWPP from Figure 8.1 includes a *release preparation* step, prior to the execution of the previously automated validation process, where all the necessary resources are predisposed according to the announcement of a new release by the TP through the EGI helpdesk. TPs are requested to provide all the necessary information about the release –including location of Linux packages, release notes and documentation links–. The EGI helpdesk is the communication link between the TP and the operators of the EGI SWPP, and thus, each release issue remains open until the candidate software product has been successfully integrated in the EGI repositories.

The next step requires that the release manager elaborates the release metadata, based on the information provided by the TP, needed to create the validation repositories for that specific version. These repositories have a temporary duration and are bound to a specific operating system and architecture. For the EGI SWPP operation, an independent issue tracking system is used that is accountable to trigger the setup of those validation repositories. This step is automatically performed as a result of attaching the corresponding metadata file to the issue.

The EGI SWPP issue tracking system provides a handful of status, sequentially going through `Unverified`, `In verification` and `Stage Rollout`. At the unverified and stage-rollout stages, an ad hoc repository is created in order for the validation team and the candidate resource providers, respectively, to test the software. Only when those activities are successfully completed, the

internal issue can be closed and thus notify back to the TP through the EGI helpdesk.

8.3.2 Statement of the problem

The formerly described process is a routine exercise triggered every time a new release is required for any of the products of the EGI software stack. In addition, as explained in the previous section, since an individual internal issue is created per each supported Linux operating system, ***the number of times this process is being executed gets increased by the number of Linux distributions*** supported by the given software product.

Furthermore, from the TP standpoint, the described process ***lacks transparency and hinders the traceability***, as two different helpdesks are used, one of those not publicly accessible. The EGI SWPP actors, i.e. the release manager and the validation team, are requested to update regularly the issue in the EGI helpdesk, but as the actual work is being carried out in the internal issue tracking system, this endeavour has proven cumbersome in practice, leading to a black box-like experience of the EGI SWPP by the TP.

8.3.3 Setting up a DevOps-like continuous validation process

As it happened with the first phase, automation is a fundamental ally to address the issues and bottlenecks in mechanical or repetitive tasks within the EGI SWPP. The aim at this second phase of the EGI SWPP modernisation is not only to automatize the release preparation step, but ultimately to ***involve the TP into the EGI SWPP***, moving towards a more accurate DevOps realisation since the TP is the driver of the process. Through a TP-centric approach, the noted lack of transparency of the EGI SWPP can be finally removed.

Figure 8.6 illustrates the different levels of automation among Phase 1 and Phase 2. In terms of automation, the fundamental improvements boil down to the management of the metadata and the composition and publication of the

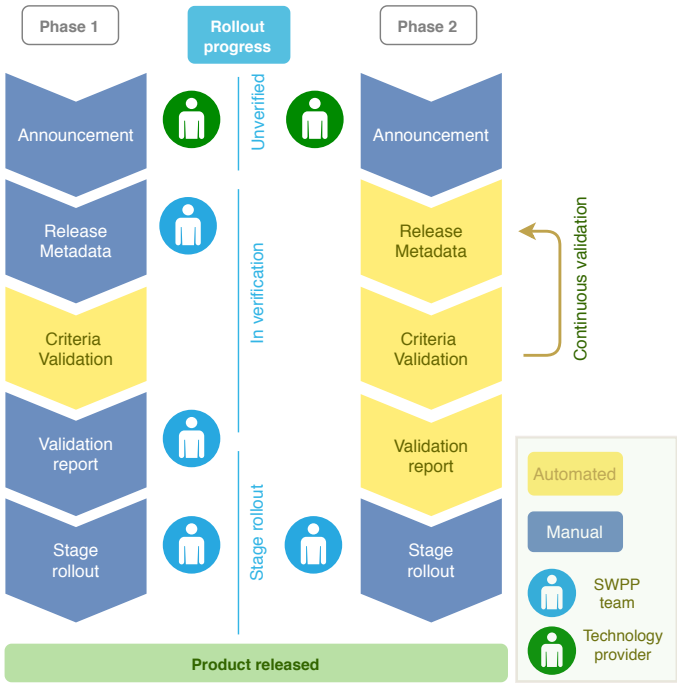

Figure 8.6: Comparison between former Phase 1 and the more advanced Phase 2. Automated steps are highlighted in yellow, while blue is used for the manual ones. Phase 2 reshapes the interaction with the TP, enabling the continuous validation of the incoming releases by means of a DevOps approach.

final validation report. Whilst the latter is a simple rendering of the results using a templating system, handling the metadata requires a deeper analysis.

The release metadata

The release metadata leverages YAML file format to describe the details of the products to be released. Each product has its own YAML description with all the information about the last version, such as the links to the documentation and artefact location or the product description. Most of this information remains unchanged between releases, as in practical terms only the *version num-*

ber, the *release notes* and the *package location* is strictly required to be provided for each release. Listing 8.2 shows a sample release metadata file for an UMD product.

```
product: "dpm"
technology_provider: "DPM"
tp_short: "DPM"
contact: "hep-service-dpm@cern.ch"
desc: "The␣Disk␣Pool␣Manager␣(DPM)␣is␣a␣lightweight␣storage␣
    solution␣for␣grid␣sites."
docs: "http://lcgdm.web.cern.ch/dpm"
releasenotes: "http://lcgdm.web.cern.ch/tags/releases"
changelog: "http://lcgdm.web.cern.ch/tags/releases"
isodate: 20192305
incremental: false
emergency: false
version: 1.12.0
capabilities: [Storage Management]
packages:
  - os: "centos7"
    arch: "x86_64"
    gpgkey: "http://repository.egi.eu/sw/production/umd/UMD-RPM
        -PGP-KEY"
    rpms:
      - "http://ftp.fi.muni.cz/pub/linux/fedora/epel/7/x86_64/
         Packages/d/d/dpm-1.12.0-2.el7.x86_64.rpm"
      - "http://ftp.fi.muni.cz/pub/linux/fedora/epel/7/x86_64/
         Packages/d/d/dpm-libs-1.12.0-2.el7.x86_64.rpm"
      - "http://ftp.fi.muni.cz/pub/linux/fedora/epel/7/x86_64/
         Packages/d/d/dpm-server-mysql-1.12.0-2.el7.x86_64.rpm"
 - os: "sl6"
    arch: "x86_64"
    gpgkey: "http://repository.egi.eu/sw/production/umd/UMD-RPM
        -PGP-KEY"
    rpms:
      - "http://ftp.fi.muni.cz/pub/linux/fedora/epel/6/x86_64/
         Packages/d/dpm-1.12.0-2.el6.x86_64.rpm"
```

```
    - "http://ftp.fi.muni.cz/pub/linux/fedora/epel/6/x86_64/
       Packages/d/dpm-libs-1.12.0-2.el6.x86_64.rpm"
    - "http://ftp.fi.muni.cz/pub/linux/fedora/epel/6/x86_64/
       Packages/d/dpm-server-mysql-1.12.0-2.el6.x86_64.rpm"
```

Listing 8.2: Release metadata in YAML format. The `packages` key lists the required artefacts needed for the deploying the given version of the software, according to the the operating system –`os` key– and architecture –`arch`–. This specific metadata would require the execution of two validation processes, one per each type of operating system.

The technology provider as the driver of the software validation

Making the metadata files available through a publicly available code repository [166] in GitHub, allows the TPs to manage the files relevant to their software products. Thus, having as a reference the DevOps approaches implemented in the DEEP and INDIGO projects, this repository is connected to the CI service, so that the criteria validation process, from Phase 1, can be triggered upon modifications on the EGI software metadata.

Opening up the criteria validation step to the TPs lays out the groundwork for a EGI's ***software validation as a service***. The announcement of a new product release now dismisses the creation of a issue in the EGI helpdesk, and instead goes directly through the metadata repository. Here, the TP updates the existing metadata with the new data for the release and, according to "Good practice VII: Use pull requests", uses a Pull Request (PR) to propose the inclusion of the new release in the EGI repositories.

The PR automatically triggers the validation of the product in Jenkins CI, by means of the execution of the `umd-verification` tool. Once finished, Jenkins reports back the results to GitHub, being reflected in the PR. If the validation fails as a result of an error on the TP side, such as malformed artefacts, wrong metadata, or detected misbehaviours, a corrected version of the release can be provided until the validation succeeds. This on-demand capacity of continuously

trigger the validation process, without the intervention of the EGI SWPP team, is what can be seen as the EGI criteria validation as a service. Note that the ***PR becomes a crucial step to guarantee that the new release only moves on to the subsequent stage rollout phase if it is approved by the EGI SWPP release manager***.

But, as it was discussed, the TP is not the only benefactor of this new DevOps approach. Unless the validation issues are caused by the EGI infrastructure, the EGI SWPP team is not involved until the validation of the new software version is successful, which dramatically reduces the workload.

Technical implementation of the continuous validation process

The creation or any subsequent modification of each PR is notified by GitHub to Jenkins, which triggers the pipeline that deals with the i) interactions with the internal tracking system –in order to change the internal ticket status–, and the ii) execution of the QC validation, reusing the implementation from Phase 1.

Figure 8.7 shows the pipeline implementation, and the series of interactions both with the metadata repository and the internal issue tracking system, i.e., the Request Tracker (RT). As it was introduced, the RT system interfaces with the EGI repository to create the required testing repositories needed for the product validation, but it has been purposely ignored in the figure for the sake of clarity.

The Jenkins pipeline validates the release metadata coming from the TP (YAML metadata) and, immediately after, converts it to the format required by RT (XML metadata). Next, the pipeline performs the Hypertext Transfer Protocol (HTTP) request in order to create the new ticket using the RT Application Programming Interface (API). This new ticket will be used to track the progress throughout the product validation. At this point, the pipeline remains in a idle status until RT reports back the successful creation of the ticket and, most importantly, the repository. With the newly created repository in place, RT delivers a HTTP POST to the URL provided by Jenkins –within the pre-

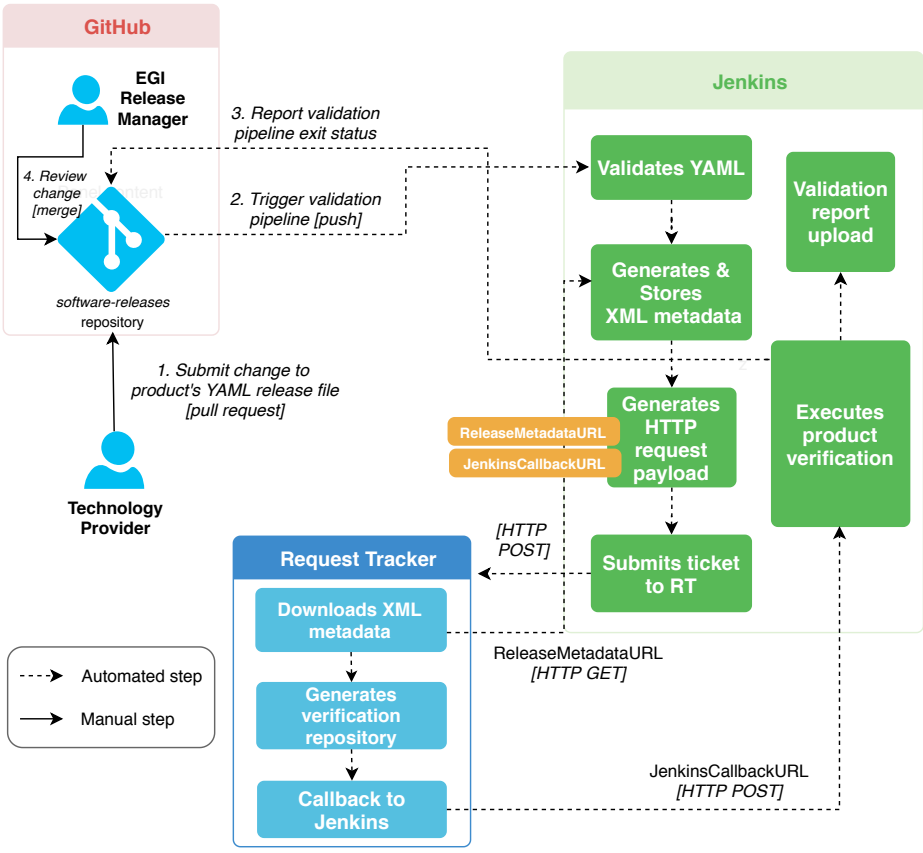

Figure 8.7: A detailed view of the workflow required for implementing the automated version of the UMD/CMD software validation process

vious HTTP request– and the pipeline continues with the execution of the EGI QC validation (Phase 1).

Whether the pipeline is successfully completed or it fails during any of the described steps, the exit status is sent back to GitHub and displayed in the PR. Hence, the TP is timely informed about the progress of the validation. At any point, but especially in the event of a failure, the TP is able to re-trigger

the validation process without the intervention of the EGI SWPP team. As soon as the validation success, the PR is subject to the approval of the release manager, who is responsible for the review [3]. Once approved, the RT ticket is set to *Stage Rollout* status, meaning that it is ready for being tested at the semi-production level in the list of candidate resource providers selected from the EGI e-Infrastructure.

At the time of writing, Phase 2 is not yet fully in production within the EGI SWPP.

8.4 Conclusion

Unlike the previous chapters, the EGI SWPP does not implement a complete DevOps Continuous Integration and Delivery (CI/CD) approach since this process is strictly committed to the validation of the software artefacts that are delivered through the UMD and CMD releases. However, the final EGI SWPP implementation, in the aftermath of the two incremental phases herein presented, does indeed implement a DevOps culture by involving the TP in the operational validation of its software products within the EGI e-Infrastructure.

As a matter of fact, automation is the driving force behind the two incremental improvements. Phase 1 focused on the EGI QC validation, which deserved prioritised intervention as it was the most time and personnel-consuming task within the EGI SWPP. The analysis of the EGI QC, in terms of automation capabilities, opened the door to the design and implementation of the `umd-verification` tool, in order to conduct the QC validation. This tool, in conjunction with CCA solutions and the Jenkins CI service, is able to deploy and test the incoming candidate software releases, with notable gains in time efficiency.

Phase 2 extends and complements the outcomes from Phase 1 by reshaping

[3]Right before the stage rollout phase, the pipeline leverages a template document to fill in the required information extracted from the EGI QC validation results to compose the validation report. This is subsequently uploaded to the EGI document database.

the previous steps, before the criteria validation, of the EGI SWPP. The fundamental shift is to make the TP the legitimate initiator of the EGI SWPP, avoiding the repetitive actions done by the release manager at the first stages of the process. Hence, the TP is accountable for the modification of the release metadata whenever a new release is due. This task is done by submitting a change to a publicly accessible code repository, which by means of a PR, triggers the criteria validation from Phase 1. The validation can be rerun as many times as needed by the TP, constituting an approximation to an EGI's validation as a service. Only when the validation is successful and approved by the release manager, the EGI SWPP goes on to the stage rollout step.

The modernisation of the EGI SWPP, towards a DevOps paradigm, resulted in the proven benefits of time and manpower efficiency, as well as the transparency and traceability improvements of the EGI SWPP from the perspective of the TP.

Act III

Mapping out the Future: Universalize and Sustain a Culture of Quality Research Software

9

Incentivize a Software Quality Culture in the European Open Science Cloud: the Software Quality Assurance as a Service

Whether in the task of developing software or when validating artefacts for the operation of a research e-Infrastructure, the previous chapters demonstrated the impact that the implementation of a Software Quality Assurance (SQA) process, according to the DevOps principles and materialised through the use of Continuous Integration and Delivery (CI/CD) pipelines, has in the final software product or service being delivered to the public. However, the design and implementation of such a SQA process is not straightforward, requiring from

expert knowledge and guidance in mainstream software engineering practices.

Throughout the past chapters, visible efforts have been promoted to support and disseminate a culture of software quality. The definition of an crowdsourced SQA baseline and the design and practical implementation –through the composition of diverse CI/CD pipeline configurations, and a general purpose library– of the requirements thereof, are visible evidence of the contribution to the establishment of this SQA culture within research software.

However, the beneficiaries of such achievements were still responding to a technical profile, with the exceptions of the specific scientific communities participants in the DEEP Hybrid-DataCloud (DEEP) and INDIGO-DataCloud (INDIGO) projects. Just as their software delivery workflows have been improved, other scientists could benefit as well drawing on the gathered expertise.

In this chapter, all the collected expertise from the realisation of the activities described in the previous chapters is condensed into the SQA-as-a-service (SQAaaS) solution. The SQAaaS, being developed at the time of writing within the framework of the EOSC-Synergy (SYNERGY) project, offers an automated solution for checking the compliance of the code with the SQA baseline from Chapter 5 "Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software", and the ability to compose on-demand CI/CD pipelines. Hence, users can exploit the benefits of having a consolidated SQA baseline assessment without the need of having prior knowledge or skills about it.

9.1 Framing the Software Quality Assurance as a Service in the European Open Science Cloud

Figure 9.1 shows the *pipelined* approach towards a continuous delivery and deployment of services in the EOSC Portal. In this view, the SQAaaS covers the verification and validation stages, including the capacity of awarding the qual-

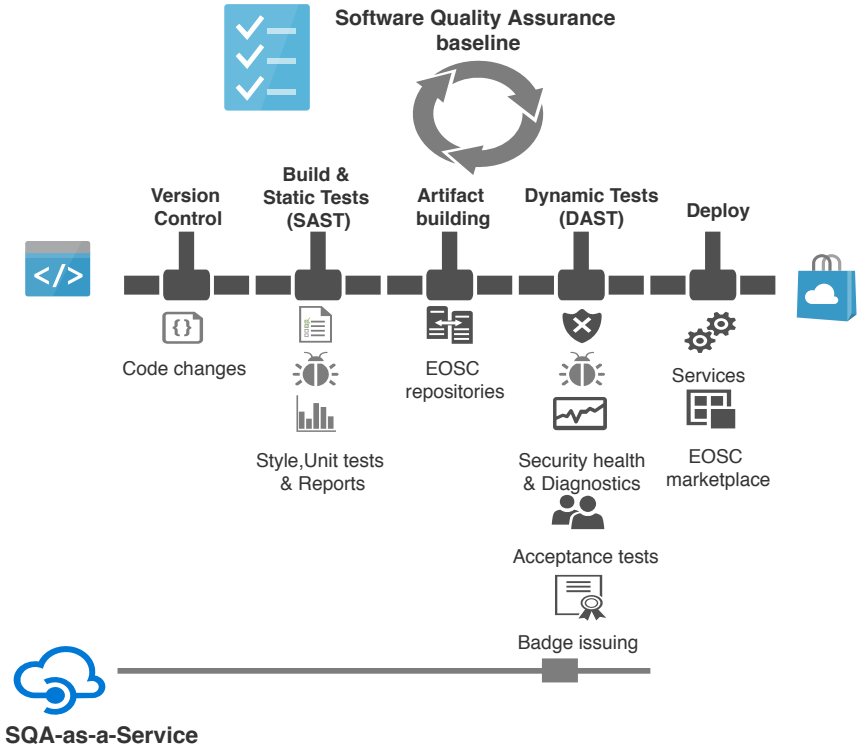

Figure 9.1: Scoping the concept of a SQAaaS in the European Open Science Cloud (EOSC). The figure represents a pipeline that performs the usual CI/CD work, verifying the source code and delivering the resultant artefacts through the EOSC repositories. The software is dynamically tested, covering security and system tests, before being delivered to the EOSC portal. As an outcome of the SQAaaS execution, quality badges are issued in order to quantify the compliance of the analysed software with regards to the SQA baseline discussed in Chapter 5.

ity of the software quality through the issuing of digital badges. The sections that follow position the SQAaaS into the EOSC context, highlighting how this service can contribute to improve the current indicators of quality and maturity in the EOSC ecosystem.

Technology Readiness Levels

In Chapter 4, it was described how the EOSC implementation was relying on the Technology Readiness Level (TRL) measurement system to quantify the maturity of a technology. However, as discussed there, TRLs leave room to ambiguous interpretations, and thus, EOSC software and services quality and maturity are usually inaccurately assessed.

With the aid of the SQA baseline, a more representative description of the quality attributes present in the software that composes those services can be then provided, contributing to a quantitative assessment of the maturity in the EOSC services. Needless to say that the SQAaaS does not displace but complement the TRL definition, by providing a better means to estimate each level.

Requirements in the SQA baseline shall be mapped with TRLs so that a clear correspondence is set. The SQAaaS could then serve as an assessment tool to endorse the assignment of TRLs to software. With a better quantification comes a greater confidence, which is not really built with the TRL definition by itself.

Award system

By highlighting the achievements, all the EOSC stakeholders –such as developers, users and funders– are aware of the quality attributes of the software. The envisaged SQAaaS would provide reports about the level of compliance that a given software product has with respect to the requirements defined in the SQA baseline. Furthermore, the quality assessment report shall be completed with the recognition of a badge to indicate the alignment with SYNERGY project standards. This outcome will contribute to the attainment of the seal of approval for EOSC or *EOSC-ready certification*, as foreseen in the 2nd High Level Expert Group (HLEG) report of the EOSC implementation [79].

9.2 Dissemination of a culture of software quality

According to the positive feedback from the experiences detailed in previous chapters, transferring this know-how and SQA culture to a broader audience is a natural step ahead, especially within those scientific environments with a more evident lack of software engineering processes.

The SQAaaS was conceived under the umbrella of the SYNERGY project, which has been designed to enable the integration of thematic services and data repositories in the EOSC. In the sections below, it will be discussed what are the principal applications of the SQAaaS.

9.2.1 Online Software Quality Assurance baseline assessment

In Chapter 5 "Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software", a set of requirements for the adequate management of the Software Development Life Cycle (SDLC) were presented, and subsequently applied to the INDIGO and DEEP projects. As described within the implementation of the SQA process of both projects, the practical implementation of the SQA baseline involved the definition of CI/CD pipelines for each software component. Throughout the years and projects, both the SQA baseline –reaching v3– and the pipelines have evolved to cover new Software Engineering (SE) methodologies and practices.

The idea of applying these such outcomes to a wider scientific computing audience was eventually shaped into the SQAaaS definition. Hence, one of its expected outputs is to assess the quality of a given code repository, according to the SQA baseline standards. The *Online SQA baseline assessment* module provides a ***comprehensive assessment of the incoming source code according to the SQA baseline*** requirements. The resultant report will contain an enumeration of the fulfilled requirements, which are then characterized

by a quality badge that determines the level of compliance with the defined SYNERGY standards.

A key aspect, particularly important when issuing the badge, is to ascertain the uniqueness of the version of the code being analysed by the SQAaaS. Based on the assumption of source code managed with a Version Control System (VCS), commit identifiers are a reference to a specific version of the code, but there is no unique identification of a code repository, and besides, these identifiers can be easily tampered. Software citation, according to "Requirement V: Make your software findable, reproducible and citable", turns out to be a better approach. The use of Persistent Identifiers (PIDs) guarantee the required uniqueness. However, this is not yet a widely used practice, and accordingly, so far the *Online SQA baseline assessment* module might eventually rely on VCS commit identifiers in the majority of cases.

9.2.2 Pipeline as a service

Just as the outcome of the previous module is aligned to the certification of software, the *Pipeline as a Service* module covers the practical aspects of the software assessment. In this case the ***expected result is the generation of a Jenkins code pipeline***, ready to be used in a Jenkins environment.

Consequently, through the *Pipeline as a Service* ***users leverage the experience gathered throughout the series of previously discussed projects*** with regards to the implementation of CI/CD pipelines. Hence, any potential scientist involved in computational programming can readily adopt the resultant pipelines, which are ready to be added to the repository of code.

In order to eliminate common technical barriers, the *Pipeline as a Service* shall allow the composition of the stages in the pipeline following an intuitive graphical approach. All the functionalities implemented in the already described `jenkins-pipeline-library` shall be accessible in order for the user to generate a customized code pipeline. The SQAaaS shall provide as well the optional feature of automatically deploying the resultant pipeline in the SYNERGY Jenk-

ins instance, as a way for users to see the execution output of the brand new pipeline.

9.3 Architecture of the Software Quality Assurance as a Service

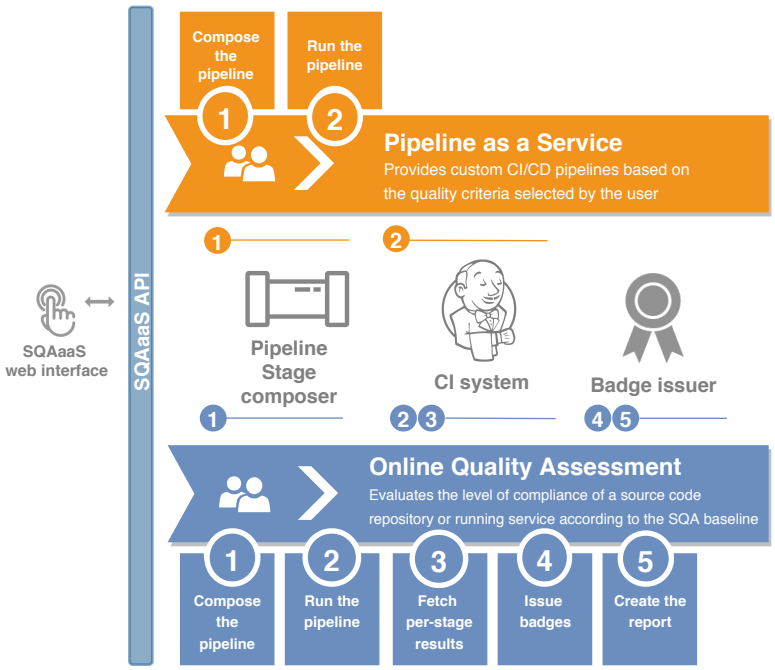

Figure 9.2: High-level overview of the SQAaaS.

Figure 9.2 shows the high-level overview of the SQAaaS architecture according to the requirements elicited in the previous section. The steps performed by the two building blocks of the SQAaaS, the *Online Quality Assessment* and the *Pipeline as a Service*, are sketched in the figure, as well as the components that carry out each action.

9.3.1 Integral components

As it can be extracted from the figure, Jenkins appears as the foundational technology used in the implementation of the SQAaaS modules. Introduced in Chapter 7 "Tailoring software to user needs: the DEEP-HybridDataCloud project", these pipelines correspond to the Jenkins code pipelines or `Jenkinsfiles`, in order to benefit from their flexibility and the capacity of being tailored programmatically. The SQAaaS trigger the execution of the pipelines, obtaining the result of each stage for the eventual display of results.

According to such results, one of the key outcomes of the SQAaaS is the issue of quality badges, according to the standards defined by SYNERGY. The SQAaaS relies on the *Badge issuer* component, which implements the Open-Badges v2.0 Specification [167], to ship digital badges –portable image files– that embed the information about the achievements and level of compliance of a given software according to the quality requirements of the SQA baseline.

Lastly, the *Pipeline as a Service* building block has the added complexity of providing an on-demand composition of the pipelines. In this case, a specific *Pipeline stage composer* component provides customized code pipelines based on the inputs obtained by the user.

9.3.2 Automated validation of the Software Quality Assurance baseline requirements

Whether in the event of validating the entire SQA baseline or a custom selection of the quality requirements thereof, the SQAaaS needs to extend the existing programmatic coverage, being built throughout the course of the DEEP and INDIGO projects.

In Chapter 8 "Software validation in the European Grid Infrastructure", a study was carried out to identify the quality requirements from the EGI Quality Criteria (EGI QC) criteria suitable to be checked programmaticaly. It was a requirement to prove the extent to which the `umd-verification` was able to validate such criteria. Now, the implementation of the SQAaaS requires a

similar approach.

Tables 9.1 and 9.2 compile all the criteria from the SQA baseline, identifying both the viability of automation and the means of verification. For some of them, the viability is aligned with the particular features that the hosting technology or platform provides. In the specific case of code management and Verification and Validation (V&V) requirements, the SQAaaS considers GitHub as the reference platform, as it has been done throughout the text.

For some criteria, the viability of automation can be partially achieved, while some requirements need additional feedback from the user in order to be assessed. However, as it can be seen in the table, current technology and tools allow to programmatically check most of the criteria included in the SQA baseline.

9.3.3 Implementation of the workflows

The interactions between the main components of the SQAaaS, already identified in Section 9.3.1, are depicted in Figure 9.3. This figure shows a more elaborated view of the different workflows of the two SQAaaS building blocks. For the sake of simplicity, the SQAaaS frontend was excluded from the previous high-level architecture figure. Here, it is included to showcase the role of the user as the initiator of the workflows.

Hence, the web frontend allows the user to select the module, either the *Online Quality Assessment* or the *Pipeline as a Service*, and the required inputs. The user selection is then passed to the the *SQAaaS director*, which is the fundamental actor responsible of managing the subsequent steps and the interaction with the backend components by leveraging their APIs.

Online Quality Assessment workflow

As introduced in Section 9.2.1, the goal of the *Online Quality Assessment* module is to gauge the quality achievements of a software component according to the SQA baseline definition. In that section, it has been also identified the

SQA criteria	Auto	Means of verification
Code Accessibility		
Requirement IV: Make your code open and publicly available	✓	`LICENSE` file
Requirement II: Use public forges to distribute your work	✓	Code repository URL
Requirement V: Make your software findable, reproducible and citable	✓	`CITATION.json` / `codemeta.json` file
Requirement III: Make clear your contribution policy	✓	`CONTRIBUTION` file
Code Management		
Requirement I: Create versions of the source code	✓	Code repository URL
Good practice I: Maintain a clean history of changes	✓	VCS history
Good practice II: Use a branching strategy to separate your development and production versions	✓	`GET /repos/:owner/:repo/branches`
Good practice VIII: Protect your long-term branches from direct modifications	✓	`GET ../protection`
Good practice III: Maintain your long–term support versions	✓	`GET /repos/:owner/:repo/branches`
Good practice IV: Use an unambiguous naming convention for branches	✓	`GET /repos/:owner/:repo/branches`
Good practice V: Use issue tracking	✓	`GET /repos/:owner/:repo/labels`
Good practice VII: Use pull requests	✓	`GET /repos/:owner/:repo/pulls`
Good practice VI: Use semantic versioning for your releases	✓	`GET /repos/:owner/:repo/releases`

Table 9.1: Automation capabilities and means of verification for Code management and Code accessibility categories from the SQA baseline. URL endpoints correspond to GitHub Application Programming Interface (API).

SQA criteria	Auto	Means of verification
Code Verification and Validation		
Requirement VI: Test the individual units of the code	✓	Run tests (input needed)
Requirement VII: Address functional requirements	✓	Run tests (input needed)
Requirement VIII: Check the level of integration and interconnection with coupled components	✓	Run tests (input needed)
Requirement IX: Ensure new changes do not jeopardize the operation of software's existing features	Partial	Run tests (input needed)
Requirement X: Adhere your code to a code style standard	✓	Run de-facto standards
Requirement XI: Assess the security on your software	Partial	Run common SAST linters
Requirement XII: Broadening the perspective with peer reviews of the code	✓	GitHub's Protection API
Software Adoption		
Requirement XIII: Comprehensive documentation	Partial	`README` file, build documentation
Good practice X: Ease the deployment of your software	✓	Execution of Ansible role / Puppet module

Table 9.2: Automation capabilities and means of verification for Code V&V and Software adoption categories from the SQA baseline.

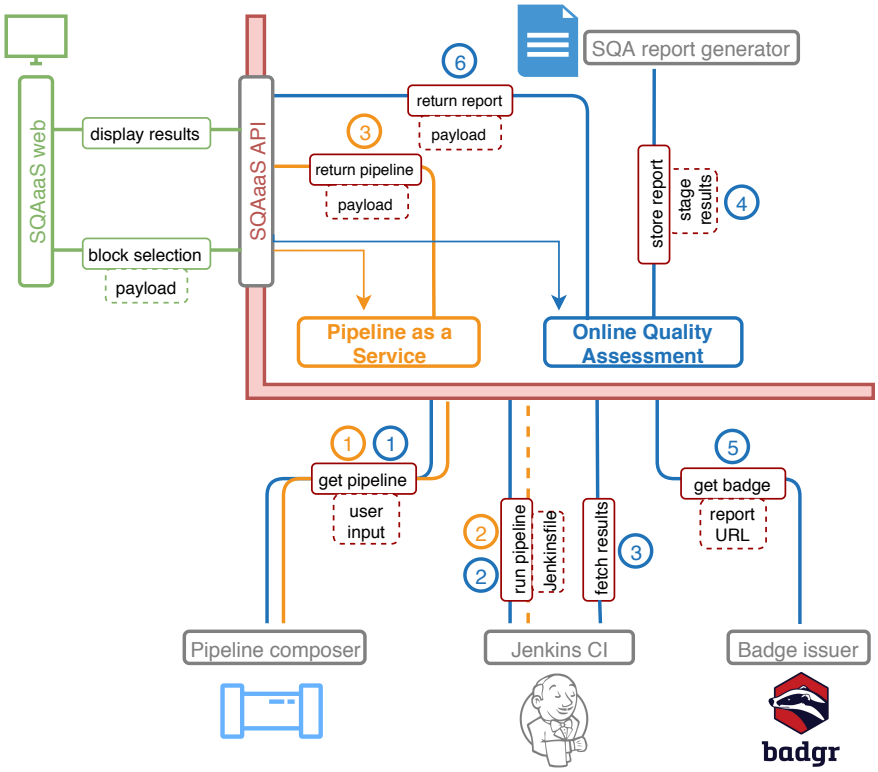

Figure 9.3: Detailed view of the SQAaaS operation. The figure details the steps required to operate each of the building blocks, i.e., the *Online Quality Assurance* (in blue) and the *Pipeline as Code* (in dark orange).

means of verification of all the requirements and practices set out therein. Based on this knowledge, the final report showcasing the level of compliance of a code repository with the SQA baseline, primary output of this module, can be then obtained.

By design, the SQAaaS operation leverages Jenkins to run the workloads. On the basis of the results obtained in Chapter 7, code pipelines are used to represent such workloads, which are generated by means of the *Pipeline composer*.

This component uses a template system to create the code pipelines according to the user preferences. Based on the needs of the *Online Quality Assessment* module, where the pipelines are pre-composed according to the programming language, the work of the *Pipeline Composer* is reduced to the selection of the appropriate pipeline according to the user input.

As in the case of the code pipelines elaborated during DEEP project, the `jenkins-pipeline-library` is used to perform the work within each pipeline stage. As it was primarily oriented to CI/CD checks, the last stable version of the library, `v1.4.1` [145], does not cover the complete set of requirements from the *Code Accessibility* and *Code Management* categories listed in Tables 9.1 and 9.2. As it was mentioned in section 9.3.2, the currently identified means of verification for those category of requirements are tied to GitHub API capabilities. Subsequent improvements might be focused on the support of additional platforms.

According to what it was discussed, the *SQAaaS director* will trigger the pipeline composition based on the provided language. Once the pipeline is generated by the *Pipeline Composer*, the director uses the Jenkins API to create a job to run the pipeline. According to the exit status of the pipeline, the director either notifies back to the user the unsuccessful execution or, in the event of successful termination, the results are then analysed by isolating each stage.

As the pipeline was earlier composed according to the `stage:SQA requirement` mapping, the report generation is quite straightforward. The final step is to issue the corresponding digital badge, according to the results obtained, that embeds the link to the resultant SQA report. Three badge categories are foreseen. As Table 9.3 showcases, the first-level badge is used for software that did not fulfill the mandatory requirements of the SQA baseline. The second and third-level represent, respectively, the accomplishment of the minimum requirements and the additional good practices.

SYNERGY badge	SQA baseline criterion type
Bronze	Not all the `Requirement` conventions fulfilled
Silver	All the `Requirement` conventions fulfilled
Gold	`Silver` badge + all or some of the `Good Practice` conventions fulfilled

Table 9.3: The three-level badges of software quality issued by SYNERGY project

Pipeline as Code workflow

The *Pipeline as Code* module follows a simpler approach as it does not need both the report creation and badge issuing capabilities. Optionally, it might not even require any interaction with the Jenkins service, but it is good practice to provide a means for the user to check the execution of the custom pipeline before adding it to the source code repository. However, in contrast, the *Pipeline as Code* module demands the dynamic composition of the pipelines, so no specific pre-composed templates are allowed here.

The resultant pipeline is rendered according to the user expectations, which implies that the *SQAaaS director* passes the user-selected criteria to the *Pipeline Composer*. In here, the templating system should consume this input data, and parameterize the stages according to them. The available features at the disposal of the SQAaaS user consist not only in the criteria that matches the SQA baseline definition, but also the remaining functionality already implemented in the `jenkins-pipeline-library`, such as the software delivery and notifications capabilities depicted in section 7.2.

9.4 Conclusion

Whilst the outcomes from the projects described in the previous chapters had limited outreach capacity, the SQAaaS solution, currently being developed within the framework of the ongoing SYNERGY project, should provide the definitive push to disseminate, to a wider audience within the EOSC, the gath-

ered expertise and insights obtained from the implementation and operation of the EGI Software Provisioning Process (EGI SWPP), DEEP and INDIGO projects.

The SQAaaS solution harnesses such experience and delivers it in two fundamental ways. On the one hand, the assessment of the software in accordance with the SQA baseline, as described in Chapter 5 "Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software". The availability of such a tool is not only relevant for any given software product, but in particular to those involved in the EOSC, as a way to improve the maturity assessment provided by the TRL system. Hence, the SQAaaS, through its *Online Quality Assessment* building block, provides a full-fledged report reflecting the level of compliance of a particular version of the software with respect to each requirement and good practice defined in the SQA baseline. Based on this result, a digital badge is issued which represents the SYNERGY's software quality standards.

On the other hand, the SQAaaS provides an on-demand composition of the code pipelines. The *Pipeline as Code* module adds to the SQA baseline criteria the set of software delivery and notification capabilities implemented throughout DEEP project in Chapter 7 "Tailoring software to user needs: the DEEP-HybridDataCloud project". As a result, any computer scientist can leverage the SQAaaS solution to compose customized pipelines and use them to implement their own CI/CD environment.

In result, the SQAaaS promotes the quality of the software at different levels. The ***pipelines covering the SQA baseline criteria stand for a fairly complete realisation of quality software***, in terms of adoption, code accessibility, management, verification and validation. Besides, the ***digital badges provide a categorization of the quality of the software***, closely aligned with the so-called *EOSC-ready certification* suggested in the EOSC 2nd HLEG report from Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud".

The ***customized pipelines can be seen as a first approach to the***

adoption of software quality practices in less-aware scientific computing environments. The SQAaaS decouples the need of professional assistance, lowering the barriers of deploying DevOps CI/CD approaches, and thus, augmenting the outreach capacity of adherence to software quality practices in the long tail of science.

Conclusions

Conclusions

9.5 Summary and Contributions

This dissertation presents a manifesto that fosters a culture of quality in the production of research software. The Open Science paradigm provides the perfect context for the establishment and commitment to such a culture that recognizes the software as a legitimate actor within the computational research. Hence, this work builds on the first steps taken towards the implementation of the EOSC, targeted to offer valuable research services for European scientists. The usability of such services is key to achieve this goal, but usability is heavily dependant on the quality of the service, not only in terms of stability and reliability, but also with regards to the functional suitability –or fit-for-purpose– that is required to meet the researcher's expectations.

Coming from an of e-Infrastructure background, where a continuum of state-of-the-art solutions have been developed and subsequently offered to a plethora of scientific communities over the years, this work builds on this expertise to demonstrate the impact that the quality of software has on the usability of the prospective service. In particular, this work is organised as a story of three sequential acts, that goes through the definition, implementation, and dissemination of a SQA process.

In particular, the major contributions laid out throughout this dissertation can be summarized as follows:

- The ***formulation of a SQA baseline to conduct the research***

SDLC. Although the original criteria were initially envisaged for guiding the SDLC within European research e-Infrastructure development projects, the SQA baseline eventually evolved to embrace a comprehensive set of software coding, testing, maintenance and uptake practices for widespread use within the global academic environment, within reach of any individual computer scientist or scientific research community. The SQA baseline has been publicly disclosed, open to external collaboration, aiming at establishing a sustainable path for collective knowledge building and transfer that consolidates it as a reference point for the development of future research software efforts.

In the context of this dissertation, the SQA baseline serves as the foundation for establishing a systematic estimation of the quality of software in the services delivered through the EOSC. The adoption of the SQA baseline by the EOSC would contribute to balance out the uneven treatment of software, with respect to data, that prevails in the ongoing EOSC roadmap, in order to address the shortcomings, in terms of quality and maturity assessment, currently existing in the service onboarding process.

- The ***implementation of a SQA process, governed by the requirements defined in the SQA baseline, that builds on the DevOps culture to develop software with an emphasis on the prospective operational performance in the target European e-Infrastructures that are contributing to the EOSC***. The initial SQA process was designed and materialised over the course of the INDIGO project, and subsequently refined during the ongoing DEEP project. Automation is the catalyst for driving the SQA process, and key enabler for the setup of a CI/CD infrastructure, where the software, in compliance with the SQA baseline criteria, is frequently delivered to be evaluated by the end users, primarily the research communities involved in the projects.

  The essential outcome of such SQA infrastructure is the composition of code pipelines, which leverage the so-called Jenkins Pipeline-as-Code tech-

nology, for each software component. The CI/CD pipelines drive the execution of the SQA baseline requirements for each modification done in the source code. To this end, the development of a Jenkins pipeline library ensures the proper fulfillment of those requirements by implementing the required methods. As a result of this coding approach, both the CI/CD pipelines and the aforementioned library are reusable outside the SQA infrastructure, thus preserving the SQA practices and know-how beyond the lifespan of the individual, commonly short-term, projects.

- The ***demonstration that the former DevOps approach is not only applicable to the –usually highly technological skilled– software engineers that develop software solutions for the operation of the e-Infrastructures, but it can be also tailored to the needs of the computer scientists***. First prototyped under INDIGO and successively developed during the development of DEEP, the pipelines are driving the development and delivery of the research applications, such as the discussed Machine Learning (ML) applications life-cycle.

- The ***application of automation beyond the bounds of usual DevOps practice in order to manage the on-demand delivery of ML applications through the DEEP Open Catalogue and the readiness for ML inference through the DEEP as a Service (DEEPaaS) endpoint***.

- The ***modernization of the EGI SWPP within the EGI Federation***, key e-Infrastructure in the EOSC implementation, to optimize the reliability of the software being delivered through the two official EGI middleware distributions: Unified Middleware Distribution (UMD) and Cloud Middleware Distribution (CMD).. The overhaul of the process included the gradual adoption of automated practices for the effective realisation of the EGI QC validation process, and subsequently, the implementation of a DevOps-like approach to achieve the continuous validation of incoming software releases, where the technology provider can autonomously trigger

the EGI QC validation process before the release can proceed to the next stages within the EGI SWPP.

- The ***outline of a SQAaaS solution, to be developed as part of the SYNERGY project, to promote and sustain a culture of quality in the research software development within the EOSC ecosystem***. On the one hand, the SQAaaS leverages the requirements set out in the SQA baseline criteria to tackle automated assessments of the quality of research software, recognising the achievements through digital badges, subject to SYNERGY standards, which set the stage for a prospective EOSC-ready seal. On the other hand, the SQAaaS provides the capacity to compose customized, ready-to-use code pipelines in order to drive the development and delivery of the software.

9.5.1 Role of the author in the reviewed research projects

All the aforementioned contributions correspond to the work led by the author of this dissertation as a result of his involvement in the operation of the SQA processes within the INDIGO, DEEP, SYNERGY projects and the European Grid Infrastructure (EGI) e-Infrastructure. Needless to say that the described work could not be completely achieved by an individual, only through joint endeavour, especially with regards to the operational part. The colleagues participating in this work are explicitly appointed in the thesis' acknowledgements.

9.6 Publications

Diverse scientific journal publications and international workshop contributions have emerged as a result of the work presented in this thesis. In particular:

- Peer-reviewed journals (main author)
  - Pablo Orviz Fernández, Mario David, Doina Cristina Duma, Elisabetta Ronchieri, Jorge Gomes, and Davide Salomoni. "Software

Quality Assurance in INDIGO-DataCloud project: a converging evolution of software engineering practices to support European Research e-Infrastructures". In: ***Journal of Grid Computing*** 18.1 (2020), pages 81–98. DOI: 10.1007/s10723-020-09509-z

- Pablo Orviz Fernández, Joao Pina, Álvaro López García, Isabel Campos Plasencia, Mario David, and Jorge Gomes. "umd-verification: Automation of Software Validation for the EGI Federated Infrastructure". In: ***Journal of Grid Computing*** 16.4 (2018). Quartile: Q1, JIF Percentile: 76.452, pages 683–696. DOI: 10.1007/s10723-018-9454-2

- Peer-reviewed journals (substantial contributions)

  - Álvaro López García ***et al.*** "A Cloud-Based Framework for Machine Learning Workloads and Applications". In: ***IEEE Access*** 8 (2020), pages 18681–18692. DOI: 10.1109/ACCESS.2020.2964386

  - Davide Salomoni, Isabel Campos, Luciano Gaido, J Marco de Lucas, P Solagna, Jorge Gomes, Ludek Matyska, P Fuhrman, Marcus Hardt, Giacinto Donvito, et al. "Indigo-datacloud: a platform to facilitate seamless access to e-infrastructures". In: ***Journal of Grid Computing*** 16.3 (2018). Quartile: Q1, JIF Percentile: 76.452, pages 381–408. DOI: 10.1007/s10723-018-9453-3

- International Workshops

  - Pablo Orviz Fernández, Mario David, and Cristina Duma. ***Baseline criteria for achieving software quality within the European research ecosystem***. Workshop. 10th Iberian Grid Computing Conference – IBERGRID 2019: University of Santiago (Santiago de Compostela, Spain), Sept. 23–26, 2019

  - Joao Pina and Pablo Orviz Fernández. ***Best practices for service deployment and interoperability checks***. Workshop. EOSC-

hub Technical Roadmap Workshop: EGI Foundation (Amsterdam, Netherlands), June 25–27, 2019

- Mario David and Pablo Orviz Fernández. ***EOSC, FAIR & Software***. Workshop. Workshop on Sustainable Software Sustainability 2019 (WOSSS19): Data Archiving, Networked Services (DANS), the Software Sustainability Institute (SSI), and the Netherlands eScience Centre (The Hague, Netherlands), Apr. 23–26, 2019
- Mikael Trellet, Pablo Orviz Fernández, and Alexandre M.J.J. Bonvin. ***DevOps adoption in scientific applications: DisVis and PowerFit cases***. Workshop. International Symposium on Grids & Clouds – ISGC 2018: Academia Sinica Grid Computing Centre (Taipei, Taiwan), Mar. 16–23, 2018

- Other relevant publications
  - Pablo Orviz Fernández, Álvaro López García, Doina Cristina Duma, Mario David, Jorge Gomes, and Giacinto Donvito. "A set of common software quality assurance baseline criteria for research projects". In: (2017). URL: http://hdl.handle.net/10261/160086

- Software
  - [Software Release] Pablo Orviz Fernández and Enol Fernández, ***egi-qc/umd-verification*** version 1.0, Apr. 2020. LIC: Apache 2.0. DOI: 10.5281/zenodo.3747669
  - [Software Release] Pablo Orviz Fernández, ***indigo-dc/jenkins-pipeline-library*** version 1.4.1, Apr. 2020. LIC: Apache 2.0. DOI: 10.5281/zenodo.3748914
  - [Software Release] Pablo Orviz Fernández, ***deephdc/schema4deep*** version 1.0, Feb. 2020. LIC: Apache 2.0. DOI: 10.5281/zenodo.3690697

9.7 Future Work and Perspective

Building on the main contributions summarized in the previous section, the following two main milestones are envisaged in the immediate future.

Service quality baseline

As discussed in Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud", the TRL scale is currently the de-facto standard for measuring the maturity of the technologies within the Horizon 2020 work programmes, and as a result, the reference system for the EOSC services. According to the hypothesis of this study, and adequate development of the underlying software is of definitive importance to deliver reliable and fit-for-purpose EOSC services. Chapter 5 "Wrapping-up: The definition of a Software Quality Assurance baseline for Research Software" presented concrete and comprehensive guidelines for accomplishing this objective, subsequently implemented in diverse EC-funded projects, and ultimately being offered as a service, through the SQAaaS solution, as a means to provide a tool to automatically assess and award the software of the EOSC services.

Hence, the quality and maturity of the technology can be adequately accomplished and measured using the tools and processes implemented as part of this thesis, complementing to a large extent the TRL system. However, there is still room for improvement in what regards to the assessment of the service operation, once being delivered by following the DevOps processes herein presented. Following a similar approach by compiling criteria for a ***service quality baseline***, can help to ensure the adequate performance of the service at runtime, seen here as the specific instance that is running the underlying quality-assured software. Criteria that shall be covered ranges from monitoring to customer and infrastructure-oriented arrangements, such as service and operational level agreements.

EOSC certification

In Chapter 9 "Incentivize a Software Quality Culture in the European Open Science Cloud: the Software Quality Assurance as a Service", the architecture of a prospective SQAaaS solution is presented. The SQAaaS provides an awarding mechanism, based on the issuing of digital badges according to the OpenBadges specification, of the quality achievements compiled in the SQA baseline. The available badges are defined according the SYNERGY standards, currently comprised by a three-level scale.

As indicated in Chapter 4 "Software Quality to drive the delivery of services in the European Open Science Cloud", the 2nd HLEG report, elaborated as part of the EOSC consultation process, fostered the establishment of an ***EOSC-ready certification system*** in order to "promote the long-term sustainability of the EOSC operation and give credit to innovative software developments". Consequently, the SQAaaS is filling this gap, currently not being considered in the preliminary steps of the EOSC implementation.

The adoption of the SQAaaS solution by the EOSC shall be considered not only as a means to certify and award software, but also to provide guidance and regulation for prospective research software development and maintenance efforts through the SQA baseline criteria. This latter aspect was already covered for the research data through the Findability, Accessibility, Interoperability and Re-usable (FAIR) principles, now the software shall take its rightful place.

Bibliography

[1] ***UNESCO Takes the Lead in Developing a New Global Standard-setting Instrument on Open Science***. 2020. URL: https://en.unesco.org/news/unesco-takes-lead-developing-new-global-standard-setting-instrument-open-science.

[2] Technology OECD Science and Industry Policy Papers. ""Making open science a reality", OECD Science". In: 25 (2015). DOI: 10.1787/5jrs2f963zs1-en.

[3] Carlos Moedas. "Open Innovation, Open Science and Open to the World–A Vision for Europe". In: ***Luxembourg: Publications Office of the European Union*** (2016).

[4] Brian Nosek et al. "Transparency and openness promotion (TOP) guidelines". In: (2016).

[5] Benedikt Fecher and Sascha Friesike. "Open science: one term, five schools of thought". In: ***Opening science***. Springer, 2014, pages 17–47. DOI: 10.1007/978-3-319-00026-8_2.

[6] Rubén Vicente-Sáez and Clara Martínez-Fuentes. "Open Science now: A systematic literature review for an integrated definition". In: ***Journal of business research*** 88 (2018), pages 428–436. DOI: 10.1016/j.jbusres.2017.12.043.

[7] ***UNESCO Recommendation on Open Science***. 2020. URL: https://en.unesco.org/science-sustainable-future/open-science/consultation.

[8] Luis Ibáñez, Rick Avila, and Stephen Aylward. "Open source and open science: how it is changing the medical imaging community". In: ***3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro, 2006.*** IEEE. 2006, pages 690–693. DOI: 10.1109/ISBI.2006.1625010.

[9] Paola Masuzzo and Lennart Martens. ***Do you speak open science? Resources and tips to learn the language***. Technical report. PeerJ Preprints, 2017.

[10] Mick Watson. "When will 'open science' become simply 'science'?" In: ***Genome biology*** 16.1 (2015), pages 1–3. DOI: 10.1186/s13059-015-0669-2.

[11] Directory of Open Access Journals. ***Directory of Open Access Journals***. 2020. URL: https://doaj.org/.

[12] Bo-Christer Björk and David Solomon. "Open access versus subscription journals: a comparison of scientific impact". In: ***BMC medicine*** 10.1 (2012), page 73. DOI: 10.1186/1741-7015-10-73.

[13] European Commission. "Guidelines to the Rules on Open Access to Scientific Publications and Open Access to Research Data in Horizon 2020". In: (2017).

[14] Eric Raymond. "The cathedral and the bazaar". In: ***Knowledge, Technology & Policy*** 12.3 (1999), pages 23–49. DOI: 10.1007/s12130-999-1026-0.

[15] Sourceforge. ***The Complete Open-Source and Business Software Platform***. 2020. URL: https://sourceforge.net/.

[16] GitHub. ***GitHub***. 2020. URL: https://github.com/.

[17] Xiaoli Chen et al. “Open is not enough”. In: ***Nature Physics*** 15.2 (2019), pages 113–119. DOI: 10.1038/s41567-018-0342-2.

[18] Christine Borgman. “The conundrum of sharing research data”. In: ***Journal of the American Society for Information Science and Technology*** 63.6 (2012), pages 1059–1078. DOI: 10.1002/asi.22634.

[19] Virginia Gewin. “Data sharing: An open mind on open data”. In: ***Nature*** 529.7584 (2016), pages 117–119. DOI: 10.1038/nj7584-117a.

[20] Erin McKiernan et al. “Point of view: How open science helps researchers succeed”. In: ***Elife*** 5 (2016), e16800. DOI: 10.7554/eLife.16800.

[21] Mark Wilkinson et al. “The FAIR Guiding Principles for scientific data management and stewardship”. In: ***Scientific data*** 3 (2016). DOI: 10.1038/sdata.2016.18.

[22] K Vermeir, S Leonelli, A Shams Bin Tariq, et al. ***Global Access to Research Software: The Forgotten Pillar of Open Science Implementation. A Global Young Academy Report***. 2018.

[23] European Commission. “Turning FAIR into reality”. In: (2018). DOI: 10.2777/1524.

[24] Lorena Barba. “Reproducibility PI Manifesto”. In: (2012).

[25] Alyssa Goodman et al. “Ten simple rules for the care and feeding of scientific data”. In: ***PLoS computational biology*** 10.4 (2014). DOI: 10.1371/journal.pcbi.1003542.g001.

[26] European Commission. “Open Science Policy Platform Recommendations”. In: (2018).

[27] Paul Ayris et al. “Open Science and its role in universities: A roadmap for cultural change”. In: ***Leuven: League of European Research Universities (LERU) Office*** 13 (2018).

[28] Carole Goble. “Better software, better research”. In: ***IEEE Internet Computing*** 18.5 (2014), pages 4–8. DOI: 10.1109/MIC.2014.88.

[29] Judith Segal and Chris Morris. “Developing scientific software”. In: ***IEEE software*** 25.4 (2008), pages 18–20. DOI: 10.1109/MS.2008.85.

[30] Udit Nangia and Daniel Katz. “Understanding software in research: Initial results from examining Nature and a call for collaboration”. In: ***2017 IEEE 13th International Conference on e-Science (e-Science)***. IEEE. 2017, pages 486–487. DOI: 10.1109/eScience.2017.78.

[31] Simon Hettrick et al. “UK research software survey 2014”. In: (2014). DOI: 10.5281/zenodo.14809.

[32] Udit Nangia, Daniel Katz, et al. “Track 1 paper: surveying the US National Postdoctoral Association regarding software use and training in research”. In: ***Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE 5.1)***. 2017. DOI: 10.5281/zenodo.814103.

[33] Arne Johanson and Wilhelm Hasselbring. “Software engineering for computational science: Past, present, future”. In: ***Computing in Science & Engineering*** (2018). DOI: 10.1109/MCSE.2018.021651343.

[34] Rob Baxter et al. “The research software engineer”. In: ***Digital Research Conference, Oxford***. 2012, pages 1–3.

[35] Daniel Katz et al. “Research Software Development & Management in Universities: Case Studies from Manchester’s RSDS Group, Illinois’ NCSA and Notre Dame’s CRC”. In: ***2019 IEEE/ACM 14th International Workshop on Software Engineering for Science (SE4Science)*** (2019). DOI: 10.1109/se4science.2019.00009.

[36] Hooman Hoodat and Hassan Rashidi. “Classification and analysis of risks in software engineering”. In: ***World Academy of Science, Engineering and Technology*** 56.32 (2009), pages 446–452.

[37] Omar Badreddin, Wahab Hamou-Lhadj, and Swapnil Chauhan. “Susereum: towards a reward structure for sustainable scientific research software”. In: ***2019 IEEE/ACM 14th International Workshop***

on Software Engineering for Science (SE4Science). IEEE. 2019, pages 51–54. DOI: 10.1109/SE4Science.2019.00015.

[38] Steve Easterbrook. "Open code for open science?" In: ***Nature Geoscience*** 7.11 (2014), page 779. DOI: 10.1038/ngeo2283.

[39] Peter Ivie and Douglas Thain. "Reproducibility in scientific computing". In: ***ACM Computing Surveys (CSUR)*** 51.3 (2018), pages 1–36. DOI: 10.1145/3186266.

[40] Jörg Fehr et al. "Best practices for replicability, reproducibility and reusability of computer-based experiments exemplified by model reduction software". In: ***arXiv preprint arXiv:1607.01191*** (2016). DOI: 10.3934/Math.2016.3.261.

[41] Pierre Bourque, Richard Fairley, et al. ***Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0***. IEEE Computer Society Press, 2014. DOI: 10.1109/52.805471.

[42] Stephen Crouch et al. "The Software Sustainability Institute: changing research software attitudes and practices". In: ***Computing in Science & Engineering*** 15.6 (2013), pages 74–80. DOI: 10.1109/MCSE.2013.133.

[43] Caroline Jay et al. "Theory-Software Translation: Research Challenges and Future Directions". In: ***arXiv preprint arXiv:1910.09902*** (2019). URL: https://arxiv.org/abs/1910.09902.

[44] William Wong. ***A management overview of software reuse***. US Department of Commerce, National Bureau of Standards, 1986. DOI: 10.6028/NBS.SP.500-142.

[45] Mike Jackson, Steve Crouch, and Rob Baxter. "Software evaluation: criteria-based assessment". In: ***Software Sustainability Institute*** (2011).

[46] José Miguel, David Mauricio, and Glen Rodríguez. “A review of software quality models for the evaluation of software products”. In: ***arXiv preprint arXiv:1412.2977*** (2014). DOI: 10.5121/ijsea.2014.5603.

[47] ISO Central Secretary. ***Systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — Measurement of quality in use***. en. Standard ISO/IEC 25022:2016. International Organization for Standardization, 2016. URL: https://www.iso.org/standard/35746.html.

[48] Michael Woelfle, Piero Olliaro, and Matthew Todd. “Open science is a research accelerator”. In: ***Nature Chemistry*** 3.10 (2011), page 745. DOI: 10.1038/nchem.1149.

[49] Il-Horn Hann et al. “Why do developers contribute to open source projects? First evidence of economic incentives”. In: ***2nd workshop on open source software engineering, Orlando, FL***. 2002.

[50] Cristina Rossi and Andrea Bonaccorsi. “Intrinsic vs. extrinsic incentives in profit–oriented firms supplying Open Source products and services”. In: ***First Monday*** 10.5 (2005). DOI: 10.5210/fm.v10i5.1242.

[51] Guido Hertel, Sven Niedner, and Stefanie Herrmann. “Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel”. In: ***Research policy*** 32.7 (2003), pages 1159–1177. DOI: 10.1016/S0048-7333(03)00047-7.

[52] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. “Is it all lost? A study of inactive open source projects”. In: ***IFIP international conference on open source systems***. Springer. 2013, pages 61–79. DOI: 10.1007/978-3-642-38928-3_5.

[53] Eirini Kalliamvakou et al. “The promises and perils of mining GitHub”. In: ***Proceedings of the 11th working conference on mining software repositories***. 2014, pages 92–101. DOI: 10.1145/2597073.2597074.

[54] Jef Raskin. “Comments are more important than code”. In: ***Queue*** 3.2 (2005), pages 64–65. DOI: 10.1145/1053331.1053354.

[55] Kent Beck et al. “Manifesto for agile software development”. In: (2001). URL: https://agilemanifesto.org/.

[56] Cem Kaner, Jack Falk, and Hung Nguyen. ***Testing computer software***. John Wiley & Sons, 1999. DOI: 10.1002/smr.4360060306.

[57] William Perry. ***Effective methods for software testing: Includes complete guidelines, Checklists and Templates***. John Wiley & Sons, 2007.

[58] Edward Kit. ***Software testing in the real world: improving the process***. Addison-wesley, 1995.

[59] Chin-Yu Huang and Michael Lyu. “Optimal release time for software systems considering cost, testing-effort and test efficiency”. In: ***IEEE transactions on Reliability*** 54.4 (2005), pages 583–591. DOI: 10.1109/TR.2005.859230.

[60] James Bullock. “Calculating the Value of Testing From an executive’s perspective, software testing is not a capital investment in the physical plant, an acquisition, or another readily accepted business expense. A Quality Assurance Manager describes how to present testing as a business-process investment”. In: ***Software Testing and Quality Engineering*** 2 (2000), pages 56–63.

[61] Roberto Di Cosmo and Stefano Zacchiroli. “Software Heritage: Why and How to Preserve Software Source Code”. In: ***iPRES 2017: 14th International Conference on Digital Preservation***. Kyoto, Japan, Sept. 25, 2017. URL: https://www.softwareheritage.org/wp-content/uploads/2020/01/ipres-2017-swh.pdf%20https://hal.archives-ouvertes.fr/hal-01590958. published.

[62] Fernando Almeida, José Oliveira, and José Cruz. "Open standards and open source: enabling interoperability". In: ***International Journal of Software Engineering & Applications (IJSEA)*** 2.1 (2011), pages 1–11. DOI: 10.5121/ijsea.2011.2101.

[63] Michael Ryan and Louis Wheatcraft. "On the Use of the Terms Verification and Validation". In: ***INCOSE International Symposium***. Volume 27, 1. Wiley Online Library. 2017, pages 1277–1290. DOI: 10.1002/j.2334-5837.2017.00427.x.

[64] IEEE Computer Society. "IEEE Standard for System and Software Verification and Validation". In: ***IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)*** (2012), pages 1–223. DOI: 10.1109/IEEESTD.2012.6204026.

[65] Ivo Babuska and Tinsley Oden. "Verification and validation in computational engineering and science: basic concepts". In: ***Computer methods in applied mechanics and engineering*** 193.36 (2004), pages 4057–4066. DOI: 10.1016/j.cma.2004.03.002.

[66] Francesca Saglietti and Florin Pinte. "Automated unit and integration testing for component-based software systems". In: ***Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems***. ACM. 2010, page 5. DOI: 10.1145/1868433.1868440.

[67] Elfriede Dustin, Jeff Rashka, and John Paul. ***Automated software testing: introduction, management and performance***. Addison-Wesley Professional, 1999.

[68] Dudekula Mohammad Rafi et al. "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey". In: ***Proceedings of the 7th International Workshop on Automation of Software Test***. IEEE Press. 2012, pages 36–42. DOI: 10.1109/IWAST.2012.6228988.

[69] Kristian Wiklund et al. “Impediments for software test automation: A systematic literature review”. In: ***Software Testing, Verification and Reliability*** 27.8 (2017). DOI: 10.1002/stvr.1639.

[70] Ossi Taipale et al. “Trade-off between automated and manual software testing”. In: ***International Journal of System Assurance Engineering and Management*** 2.2 (2011), pages 114–125. DOI: 10.1007/s13198-011-0065-6.

[71] ***Agile Infrastructure: A Story in Three Acts***. Agile Velocity 2009. June 25, 2009. URL: https://www.slideshare.net/littleidea/agile-infrastructure-velocity-09 (visited on 02/06/2020).

[72] Youssef Bassil. “A simulation model for the waterfall software development life cycle”. In: ***arXiv preprint arXiv:1205.6904*** (2012). URL: https://arxiv.org/abs/1205.6904.

[73] Nicole Forsgren et al. ***Accelerate: State of DevOps 2019***. Technical report. 2019.

[74] ***Jenkins CI***. 2020. URL: https://jenkins.io/.

[75] ***A Digital Single Market Strategy for Europe***. European Commission. June 5, 2015. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:52015DC0192&from=EN (visited on 02/06/2020).

[76] e-IRGSP5 project. ***Guide to e-Infrastructure Requirements for European Research Infrastructures***. Technical report. e-IRG, Mar. 2017. URL: http://e-irg.eu/documents/10920/363494/2017-Supportdocument.pdf.

[77] ***Integrating and managing services for the European Open Science Cloud (EOSC-hub)***. 2020. URL: https://cordis.europa.eu/project/id/777536.

[78] Paul Ayris et al. “Realising the European Open Science Cloud”. In: (2016). DOI: 10.2777/940154.

[79] ***Prompting an EOSC in practice***. European Commission. Nov. 20, 2018. URL: https://ec.europa.eu/info/publications/prompting-eosc-practice_en (visited on 02/06/2020).

[80] ***EOSC Declaration***. European Commission. June 12, 2017. URL: https://ec.europa.eu/research/openscience/pdf/eosc_declaration.pdf (visited on 02/06/2020).

[81] ***Implementation Roadmap for the European Open Science Cloud***. European Commission. Mar. 14, 2018. URL: https://ec.europa.eu/research/openscience/pdf/swd_2018_83_f1_staff_working_paper_en.pdf (visited on 02/06/2020).

[82] ***European Open Science Cloud (EOSC) Strategic Implementation Plan***. European Commission. June 1, 2019. URL: https://ec.europa.eu/info/publications/european-open-science-cloud-eosc-strategic-implementation-plan_en (visited on 02/06/2020).

[83] ***European Open Science Cloud (EOSC) Work Plan 2019-2020***. European Commission. Aug. 1, 2019. URL: https://op.europa.eu/es/publication-detail/-/publication/3c379ccc-ee2c-11e9-a32c-01aa75ed71a1 (visited on 02/06/2020).

[84] ***Horizon 2020, Work Programme 2014–2015***. European Commission. July 22, 2014. URL: https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/main/h2020-wp1415-intro_en.pdf (visited on 02/06/2020).

[85] ITIL Official-Site. ***ITIL glossary and abbreviations. ITIL official-site***. 2011.

[86] Standards for lightweight IT service management. ***Part 0: Overview and vocabulary***. Technical report. 2016. URL: https://wiki.eosc-hub.eu/download/attachments/26413993/FitSM-0_Overview_and_vocabulary.pdf?version=1&modificationDate=1530097800882&api=v2.

[87] EOSC-hub project consortium. ***Deliverable 4.1: Operational requirements for the services in the catalogue***. Technical report. 2018. URL: https://documents.egi.eu/secure/ShowDocument?docid=3342.

[88] ***FitSM***. 2020. URL: https://apmg-international.com/product/fitsm.

[89] John Shepherdson. ***CESSDA Software Maturity Levels***. Version 3. Mar. 2019. DOI: 10.5281/zenodo.2614050.

[90] ***UNESCO and Software***. 2020. URL: http://www.unesco.org/new/en/communication-and-information/resources/news-and-in-focus-articles/in-focus-articles/2004/unesco-and-software/.

[91] ***EGI Federation***. 2020. URL: https://www.egi.eu/federation/.

[92] ***Enabling Grids for E-sciencE (EGEE)***. 2020. URL: https://cordis.europa.eu/project/id/222667.

[93] A Candiello et al. "A business model for the establishment of the European Grid Infrastructure". In: ***Journal of Physics: Conference Series***. Volume 219. 6. IOP Publishing. 2010, page 062011. DOI: 10.1088/1742-6596/219/6/062011.

[94] EGI Quality Assurance team. ***EGI Quality Criteria 7th release***. http://egi-qc.github.io/. Online; accessed April 1st, 2018. 2018.

[95] ***UMD product ID cards***. 2020. URL: https://wiki.egi.eu/wiki/UMD_products_ID_cards.

[96] ***CMD products***. 2020. URL: https://wiki.egi.eu/wiki/EGI_Cloud_Middleware_Distribution_products.

[97] Davide Salomoni et al. "Indigo-datacloud: a platform to facilitate seamless access to e-infrastructures". In: ***Journal of Grid Computing*** 16.3 (2018). Quartile: Q1, JIF Percentile: 76.452, pages 381–408. DOI: 10.1007/s10723-018-9453-3.

[98] Pablo Orviz Fernández et al. "Software Quality Assurance in INDIGO-DataCloud project: a converging evolution of software engineering practices to support European Research e-Infrastructures". In: ***Journal of Grid Computing*** 18.1 (2020), pages 81–98. DOI: 10.1007/s10723-020-09509-z.

[99] Álvaro López García ***et al.*** "A Cloud-Based Framework for Machine Learning Workloads and Applications". In: ***IEEE Access*** 8 (2020), pages 18681–18692. DOI: 10.1109/ACCESS.2020.2964386.

[100] Pablo Orviz Fernández et al. "A set of common software quality assurance baseline criteria for research projects". In: (2017). URL: http://hdl.handle.net/10261/160086.

[101] ***INtegrating Distributed data Infrastructures for Global ExplOitation (INDIGO-DataCloud)***. 2020. URL: https://cordis.europa.eu/project/id/777435.

[102] Jorge Gomes ***et al. INDIGO-DataCloud Deliverable 3.1: Initial Plan for WP3***. Technical report. INDIGO-DataCloud project, June 2015. URL: https://www.indigo-datacloud.eu/documents/initial-plan-wp3-d31.

[103] ***Designing and Enabling E-infrastructures for intensive Processing in a Hybrid DataCloud (DEEP-Hybrid-DataCloud)***. 2020. URL: https://cordis.europa.eu/project/id/777435.

[104] ***eXtreme DataCloud***. 2020. URL: https://cordis.europa.eu/project/id/777367.

[105] [Software Release] Pablo Orviz Fernández et al., ***indigo-dc/sqa-baseline*** version v3.0, Feb. 2020. LIC: CC-BY-SA-4.0. DOI: 10.5281/zenodo.3673956.

[106] Scott Bradner. ***Key words for use in RFCs to Indicate Requirement Levels***. BCP 14. RFC Editor, 1997. URL: http://www.rfc-editor.org/rfc/rfc2119.txt.

[107] Jon Loeliger and Matthew McCullough. ***Version Control with Git: Powerful tools and techniques for collaborative software development.*** " O'Reilly Media, Inc.", 2012.

[108] Nayan B Ruparelia. "The history of version control". In: ***ACM SIGSOFT Software Engineering Notes*** 35.1 (2010), pages 5–9. DOI: 10.1145/1668862.1668876.

[109] Victoria Stodden. "Reproducible research for scientific computing: Tools and strategies for changing the culture". In: ***Computing in Science & Engineering*** 14.4 (2012), page 13. DOI: 10.1109/MCSE.2012.38.

[110] Jeff Kreeftmeijer. ***Using git-flow to automate your git branching workflow.*** 2015.

[111] Tom Preston-Werner. "Semantic Versioning 2.0.0". In: ***Semantic Versioning. Available: https://semver.org/.[cited 18 Apr 2018]*** (2013). URL: https://semver.org/spec/v2.0.0.html.

[112] Valerio Cosentino, Javier Cánovas Izquierdo, and Jordi Cabot. "A systematic mapping study of software development with GitHub". In: ***IEEE Access*** 5 (2017), pages 7173–7192. DOI: 10.1109/ACCESS.2017.2682323.

[113] Laura Dabbish et al. "Social coding in GitHub: transparency and collaboration in an open software repository". In: ***Proceedings of the ACM 2012 conference on computer supported cooperative work.*** ACM. 2012, pages 1277–1286. DOI: 10.1145/2145204.2145396.

[114] Linus Nyman and Juho Lindman. "Code forking, governance and sustainability in open source software". In: ***Technology Innovation Management Review*** 3.1 (2013). DOI: 10.22215/timreview/644.

[115] Andrew Silver. "Microsoft's purchase of GitHub leaves some scientists uneasy". In: ***Nature*** 558.7710 (2018), pages 353–354. DOI: 10.1038/d41586-018-05426-0.

[116] Sam Ricketson and Jane C Ginsburg. ***International copyright and neighboring rights: the Berne convention and beyond***. Oxford University Press, 2006. URL: https://scholarship.law.columbia.edu/books/96.

[117] Andrew Morin, Jennifer Urban, and Piotr Sliz. "A quick guide to software licensing for the scientist-programmer". In: ***PLoS computational biology*** 8.7 (2012), e1002598. DOI: 10.1371/journal.pcbi.1002598.

[118] Joseph Feller, Brian Fitzgerald, et al. ***Understanding open source software development***. Addison-Wesley London, 2002.

[119] OpenStack Foundation. ***LegalIssuesFAQ***. 2020. URL: https://wiki.openstack.org/wiki/LegalIssuesFAQ#Copyright_Headers.

[120] Arfon M Smith, Daniel S Katz, and Kyle E Niemeyer. "Software citation principles". In: ***PeerJ Computer Science*** 2 (2016), e86. DOI: 10.7717/peerj-cs.86.

[121] Matthew B Jones et al. "CodeMeta: an exchange schema for software metadata". In: ***KNB Data Repository*** (2016). DOI: 10.5063/schema/codemeta-2.0.

[122] Elsevier. ***SoftwareX journal***. 2020. URL: https://www.journals.elsevier.com/softwarex.

[123] Arfon ***et al.*** Smith. "Journal of Open Source Software (JOSS): design and first-year review". In: ***PeerJ Comput. Sci.*** (2018). DOI: 10.7717/peerj-cs.147.

[124] Qian Yang, J Jenny Li, and David M Weiss. "A survey of coverage-based testing tools". In: ***The Computer Journal*** 52.5 (2009), pages 589–597. DOI: 10.1145/1138929.1138949.

[125] Ankunda R Kiremire. "The application of the pareto principle in software engineering". In: ***Consulted January*** 13 (2011), page 2016.

[126] Shin Yoo and Mark Harman. “Pareto efficient multi-objective test case selection”. In: ***Proceedings of the 2007 international symposium on Software testing and analysis***. 2007, pages 140–150. DOI: 10.1145/1273463.1273483.

[127] James Williams and Anand Dabirsiaghi. “The unfortunate reality of insecure libraries. Aspect Security”. In: ***Inc., March*** (2012).

[128] ***OWASP Secure Coding Practices Quick Reference Guide***. The Open Web Application Security Project. 2011. URL: https://www.owasp.org/images/a/aa/OWASP_SCP_Quick_Reference_Guide_SPA.pdf (visited on 2020).

[129] J Williams. “OWASP Code Review Guide”. In: ***OWASP Foundation*** 2.0 (2013). URL: https://owasp.org/www-pdf-archive/OWASP_AlphaRelease_CodeReviewGuide2.0.pdf.

[130] James D Herbsleb and Deependra Moitra. “Global software development”. In: ***IEEE software*** 18.2 (2001), pages 16–20. DOI: 10.1109/52.914732.

[131] Steven Rakitin. “Manifesto elicits cynicism”. In: ***IEEE computer*** 34.12 (2001), page 4.

[132] Read the Docs. ***Read the Docs***. 2020. URL: https://readthedocs.org/.

[133] Gitbook. ***Gitbook***. 2020. URL: https://www.gitbook.com/.

[134] [Software Module] DeHaan, Michael, ***Ansible***, Ansible Inc. / Red Hat Inc. LIC: GNU. URL: https://www.ansible.com/.

[135] [Software Module] Puppet, Puppet Enterprise. LIC: Apache 2.0. URL: https://www.puppet.com/.

[136] [Software Module] Chef, Chef Community. LIC: Apache 2.0. URL: https://www.chef.io/.

[137] [Software Module] Hykes, Solomon, ***Docker***, Docker, Inc. LIC: Apache 2.0. URL: https://www.docker.com/.

[138] OpenStack Foundation. ***OpenStack***. 2020. URL: http://www.openstack.org%20http://openstack.org.

[139] GitHub. ***GitHub API v3 — GitHub Developer Guide***. 2019. URL: https://developer.github.com/v3/.

[140] GCP Van Zundert and Alexandre MJJ Bonvin. "DisVis: quantifying and visualizing accessible interaction space of distance-restrained biomolecular complexes". In: ***Bioinformatics*** 31.19 (2015), pages 3222–3224. DOI: 10.1093/bioinformatics/btv333.

[141] Gydo Cp van Zundert and Alexandre Mjj Bonvin. "Fast and sensitive rigid-body fitting into cryo-EM density maps with PowerFit". In: ***AIMS Biophysics*** 2.2 (2015), pages 73–87. DOI: 10.3934/biophy.2015.2.73.

[142] ***indigo-dc Ansible Galaxy***. 2020. URL: https://galaxy.ansible.com/indigo-dc/.

[143] ***GitBook's indigo-dc organization***. 2020. URL: https://www.gitbook.com/@indigo-dc.

[144] Jenkins. ***Pipeline Syntax***. 2020. URL: https://jenkins.io/doc/book/pipeline/syntax/.

[145] [Software Release] Pablo Orviz Fernández, ***indigo-dc/jenkins-pipeline-library*** version 1.4.1, Apr. 2020. LIC: Apache 2.0. DOI: 10.5281/zenodo.3748914.

[146] [Software Release] Pablo Orviz Fernández, ***deephdc/schema4deep*** version 1.0, Feb. 2020. LIC: Apache 2.0. DOI: 10.5281/zenodo.3690697.

[147] JSON Schema. ***JSON Schema***. 2020. URL: https://json-schema.org/.

[148] [Software Module] Julian Berman, ***An implementation of JSON Schema validation for Python***, 2020Python Package Index. LIC: MIT. URL: https://pypi.org/project/jsonschema/.

[149] JSON Schema. ***JSON Schema Draft-07 Release Notes***. 2020. URL: https://json-schema.org/draft-07/json-schema-release-notes.html.

[150] [SOFTWARE] DEEP-Hybrid-DataCloud Consortium, ***deephdc/deep-hdc.github.io***, 2020. URL: https://github.com/deephdc/deephdc.github.io.

[151] DEEP-Hybrid-DataCloud. ***DEEP as a Service***. 2020. URL: http://deepaas.deep-hybrid-datacloud.eu/.

[152] Pablo Orviz Fernández et al. "umd-verification: Automation of Software Validation for the EGI Federated Infrastructure". In: ***Journal of Grid Computing*** 16.4 (2018). Quartile: Q1, JIF Percentile: 76.452, pages 683–696. DOI: 10.1007/s10723-018-9454-2.

[153] Erwin Laure et al. ***Programming the Grid with gLite***. Technical report. 2006. DOI: 10.12921/cmst.2006.12.01.33-45.

[154] ***EGI Unified Middleware Distribution repository***. 2020. URL: http://repository.egi.eu/sw/production/umd/.

[155] ***European Middleware Initiative (EMI)***. 2020. URL: https://cordis.europa.eu/project/id/261611/.

[156] Enol Fernández-del-Castillo, Diego Scardaci, and Álvaro López García. "The EGI federated cloud e-infrastructure". In: ***Procedia Computer Science*** 68 (2015), pages 196–205. DOI: 10.1016/j.procs.2015.09.235.

[157] ***EGI Cloud Middleware Distribution repository***. 2020. URL: http://repository.egi.eu/sw/production/cmd-os/.

[158] Mario David ***et al.*** "Validation of Grid Middleware for the European Grid Infrastructure". In: ***Journal of Grid Computing*** 12.3 (2014), pages 543–558. DOI: 10.1007/s10723-014-9301-z.

[159] [Software Release] Pablo Orviz Fernández and Enol Fernández, ***egi-qc/umd-verification*** version 1.0, Apr. 2020. lic: Apache 2.0. doi: 10.5281/zenodo.3747669.

[160] Jeff Forcier. "Fabric documentation". In: ***Published*** 26 (2018), page 2018.

[161] National Institute of Standards and Technology (NIST). "Secure Hash Standard". In: ***Federal Inf. Process. Stds. (NIST FIPS)*** (2015), pages 180–4. doi: 10.6028/NIST.FIPS.180-4.

[162] Open Grid Forum. ***GLUE Specification v. 2.*** Online; accessed April 1st, 2018. 2020. url: https://www.ogf.org/documents/GFD.147.pdf.

[163] EGI Software Provisioning team. ***EGI Quality Criteria in Ansible Galaxy***. https://galaxy.ansible.com/egi-qc/. 2020.

[164] EGI Software Provisioning team. ***EGI Quality Criteria in PuppetForge***. https://forge.puppet.com/egiqc/. 2020.

[165] EGI Software Provisioning team. ***EGI Quality Criteria in GitHub***. https://github.com/egi-qc. 2020.

[166] EGI Software Provisioning team. ***software-releases***. https://github.com/egi-qc/software-releases. 2020.

[167] IMS Global Learning Consortium et al. "Open Badges v2. 0 IMS final release". In: ***IMS Global Learning Consortium*** (2018). url: https://www.imsglobal.org/sites/default/files/Badges/OBv2p0Final/index.html.

[168] Pablo Orviz Fernández, Mario David, and Cristina Duma. ***Baseline criteria for achieving software quality within the European research ecosystem***. Workshop. 10th Iberian Grid Computing Conference – IBERGRID 2019: University of Santiago (Santiago de Compostela, Spain), Sept. 23–26, 2019.

[169] Joao Pina and Pablo Orviz Fernández. ***Best practices for service deployment and interoperability checks***. Workshop. EOSC-hub Technical Roadmap Workshop: EGI Foundation (Amsterdam, Netherlands), June 25–27, 2019.

[170] Mario David and Pablo Orviz Fernández. ***EOSC, FAIR & Software***. Workshop. Workshop on Sustainable Software Sustainability 2019 (WOSSS19): Data Archiving, Networked Services (DANS), the Software Sustainability Institute (SSI), and the Netherlands eScience Centre (The Hague, Netherlands), Apr. 23–26, 2019.

[171] Mikael Trellet, Pablo Orviz Fernández, and Alexandre M.J.J. Bonvin. ***DevOps adoption in scientific applications: DisVis and PowerFit cases***. Workshop. International Symposium on Grids & Clouds – ISGC 2018: Academia Sinica Grid Computing Centre (Taipei, Taiwan), Mar. 16–23, 2018.

[172] Greg Wilson. "Software Carpentry: lessons learned". In: ***F1000Research*** 3 (2014). DOI: 10.12688/f1000research.3-62.v2.

[173] Software Sustainability Institute. ***Manifesto***. 2020. URL: https://www.software.ac.uk/about/manifesto.

[174] Greg Wilson et al. "Best practices for scientific computing". In: ***PLoS biology*** 12.1 (2014). DOI: 10.1371/journal.pcbi.1005510.

[175] Greg Wilson et al. "Good enough practices in scientific computing". In: ***PLoS computational biology*** 13.6 (2017). DOI: 10.1371/journal.pcbi.1005510.

[176] Rafael C Jiménez et al. "Four simple recommendations to encourage best practices in research software". In: ***F1000Research*** 6 (2017). DOI: 10.12688/f1000research.11407.1.

[177] Markus List, Peter Ebert, and Felipe Albrecht. ***Ten Simple Rules for Developing Usable Software in Computational Biology***. 2017. DOI: 10.1371/journal.pcbi.1005265.

[178] Peter Sabev and Katalina Grigorova. ***A Comparative Study of GUI Automated Tools for Software Testing***. 2017. URL: https://www.thinkmind.org/download.php?articleid=softeng_2017_1_20_64068.

[179] Shin Yoo and Mark Harman. "Regression testing minimization, selection and prioritization: a survey". In: ***Software testing, verification and reliability*** 22.2 (2012), pages 67–120. DOI: 10.1002/stvr.430.

[180] Haralambos Mouratidis. ***Integrating Security and Software Engineering: Advances and Future Visions: Advances and Future Visions***. Igi Global, 2006.

[181] Andrew Stellman and Jennifer Greene. ***Learning agile: Understanding scrum, XP, lean and kanban***. O'Reilly Media, Inc., 2014.

[182] ***European Cloud Initiative - Building a competitive data and knowledge economy in Europe***. European Commission. Apr. 19, 2016. URL: https://ec.europa.eu/newsroom/dae/document.cfm?doc_id=15266 (visited on 02/06/2020).

[183] ***List of institutions endorsing the EOSC Declaration***. European Commission. June 12, 2017. URL: https://ec.europa.eu/research/openscience/pdf/list_of_institutions_endorsing_the_eosc_declaration.pdf (visited on 02/06/2020).

Appendices

A

Software Quality Assurance

A.1 A representative view of educational initiatives for research software development

Software Carpentry is an example of how a successful national training initiative turns into a worldwide volunteer effort dedicated to teach computing skills to researchers. Both through in-house workshops and online training materials, Software Carpentry's aim primarily focuses on programming, grounded in the fact that scientists are largely self-taught, but also dives into related aspects such as scripting or version control [172].

In terms of national efforts, the Software Sustainability Institute (SSI) pro-

vides since 2010 educational services for researchers in the United Kingdom emphasizing the sustainable demands of software in research. The work carried out by the SSI ranges from consultancy services for researchers to lower the barriers encountered by the software they are using, to software training through the support to the aforementioned Software Carpentry movement. SSI also funds advanced training to software research enthusiasts as an strategy to gather intelligence that can eventually be used for the SSI to develop policy to influence positive change on software research matters [42].

SSI has been the main driver for the recognition of the Research Software Engineer (RSE) role in research, being an integral part of its manifesto [173]. Manifestos are common ways to condense policies, aims and beliefs about a given field or organisation. Software in research has received contributions in form of manifestos meant to identify the key goals for its better quality, reliability, sustainability and/or reproducibility. Lorena A. Barba coined as the *Reproducibility PI Manifesto* [24] a collection of eight practical-oriented recommendations for achieving reproducibility, conveying the importance of focusing on the V&V processes while developing the code and publishing it at the paper submission time.

Research publications have also extensively covered the good practices, recommendations or rules to consider when developing software for scientific use [174, 175, 176, 177]. These publications focus on the minimal software quality requirements, usually ranging from four to ten, that a scientific shall consider when developing software. They result from the experience collected in diverse scientific domains, and are intended to conduct to a better usability of the software produced in research.

Lastly, the last cluster covers the Research Infrastructures (RIs), which are required to guarantee a high level of availability and reliability in the services they provide to scientists. Especially in the European research ecosystem, several RIs came up with criteria to assess the level of quality in the software to be deployed in such infrastructures. The EGI Federation has defined a quality criteria document [94] containing the minimum requirements that the software

shall fulfill before being distributed and deployed in the underlying federation of resource providers that conform the EGI infrastructure. The CESSDA RI for social science has defined ten maturity levels of software [89] so that minimum levels can be mandated to service providers.

A.2 Test to build trust

Testing is crucial for achieving a good enough level of quality and particularly for improving the reliability of the system. Far too often, little attention is dedicated to testing with the premise of being a time-consuming task that does not pay back the effort delivered. This is partially true since it is impractical to assess all the potential *test cases* that might fall into a specific section of the code. Therefore, such high standards shall not be the target goal, but instead strive to identify and cover the most-used functional parts of the code that optimize the rewards obtained. Commonly, this goal is not even considered and, consequently, a great amount of effort is spent in segments of the code that are not relevant from a usability perspective.

Static and dynamic testing

A practical way to put V&V into action is referring to the type of testing associated to each process. Hence, ***software verification implies the static analysis*** of the source code, through the reviews and audit processes [41]. Instead, ***software validation*** requires the software to be executable in order to be tested, so it ***is identified with the dynamic behaviour*** of the software. Consequently, static testing is different from and complementary to the dynamic testing. Only both combined can cover the testing requirements of the software.

Unit testing

Unit testing is generally considered as the first level of software testing, being a valuable tool to narrow down where bugs might present in the code. It eval-

uates possible flows –according to the test cases– in the internal design of the code through the identification and isolation of the individual units, which will be then tested separately. A *unit* is thereby commonly defined as the smallest testable piece of source code, usually mapped to classes, functions or even modules, depending on the programming context (object-oriented, procedural, etc.). Being a static analysis type of testing, it contributes to uncover source code flaws early at coding time, before building the software.

Functional testing

Unit testing by itself does not provide an estimation of the quality of the code as there is no measurement about the completeness of each unit test. Even when all the statements in the code have been covered, there is no guarantee that the test cases are complete in terms of either uncovering hidden errors in the execution or validating the functional requirements. While the former can be alleviated by identifying the missing test cases in a thorough peer code review, functional testing helps in extending the scope and the effectiveness of the test cases.

The functional test cases should tend to cover all the scenarios associated with the set of requirements outlined in the design specification, and include failure paths and boundary cases. However, they lack the user component: they are written and ran by developers in order to check that the software meets its functional specifications. The heterogeneity of such scenarios is best undertaken programmatically using diverse frameworks and/or libraries, although in some cases they might be solely carried out using manual walkthroughs. Graphical user interfaces (GUIs) are clear examples when it comes to testing functionalities that involve human interactions. Fortunately, the extending capabilities of software testing frameworks are progressively providing an automated means to testing most graphical functionalities [[178]]. One such example is the Selenium [1] suite, which automates web applications for testing purposes.

[1] https://www.seleniumhq.org/

Integration testing

Unit and functional testing circumvent the interactions with external components by using mocked objects to simulate their operation, as they were irrelevant for the purpose of such testing strategies. Integration testing is suited for the verification of the real interactions among coupled software components or parts of a system that cooperate to achieve a given functionality. Unlike unit and functional tests, integration testing exposes the software to an uncontrolled environment, where the system may behave in an unpredictable way, not completely envisaged at development time. It is then a convenient last checkpoint to conclude the testing part of the verification phase, safeguarding the system against software bugs, as ideally the integration tests are executed in an environment similar to the production one.

Regression testing

There is a shared commonality across the above-described testing strategies, which implies the successful execution of the existing test cases in the event of new features being added. Known as regression testing, it has the commitment of uncovering errors in the new features by preserving the operation of of the program's defining functionalities. Identifying those is key to alleviate the incremental growth of regression test cases, as the *retest-all* approach might lead to an impractical situation where the effort or time spent on the validation phase is unacceptable [179].

Code style

Code or programming style can be defined as the set of conventions, attached to a given programming language, aiming to govern the general practices, structure and typographical appearance of the source code. Control structure, line length, indentation or commenting are samples of style factors or characteristics are involved in the definition of a code style.

Security testing

There are two different types according to whether they are applied to the code or the program, corresponding to the static or dynamic application security testing, respectively. Education on security is the basis to avoid common unsafe code statements or being exposed to known vulnerabilities, but, as far as research software is concerned, there is a general lack of awareness and skills that is originated from the traditional gap between security and software engineering [180]

Static Application Security Testing (SAST): High severity security issues might not only have future strong risk implications –if they are not promptly detected and resolved–, but the task of solving them at later stages usually becomes increasingly tougher.

Dynamic Application Security Testing (DAST): relies on the black-box methods used by attackers to compromise the security of online applications, mainly web-based. Although there is a broad range of *abuse cases* that aim at challenging the system operation, the malicious attacks commonly follow the same pattern, and thus, DAST first and foremost builds protection against those.

Code review

Ideally, code reviewers shall have significant experience with the programming languages involved and testing background. Usually, code reviews are carried out by core members, not actually involved in the implementation of the candidate change. External or *drive-by*[2] reviews are always welcome, but candidates are not easy to find. In either case, the amount of revisions ought to be balanced; too few entails the risk of incomplete analyses, while too many might contribute to unnecessary delays in important feature or bug fix releases. Code review stage should not be a barrier for the agility of the continuous improvement of the software, and as such, ***reviewers need to reconcile desired actions with available effort***, by prioritizing them or eventually proposing

[2] As an analogy to the *drive-by* commits

the division of the change in smaller bits whenever a considerable intervention is required.

A.3 Agile software development

Unlike the traditional and rigid approaches, the agile development promotes instead a rapid adaptation to changing requirements, as a result of involving the end user deeply in the development process (user-centric approach). Several frameworks have become popular since the advent of the Agile manifesto [55], in particular Kanban and Scrum [181]. Both approaches share the common goal of delivering results in the shortest possible time, but while Kanban implements a continuous workflow, Scrum delivers work in regular batches or sprints. Accordingly, Kanban is most suitable for software projects with great demands of flexibility.

To this end, agile frameworks follow an iterative and incremental procedure where a Minimum Viable Product (MVP) is delivered at each iteration. The MVP shall be operational enough to be exploited by the user and, thus, obtained useful feedback to be applied in the incoming iteration. Consequently, agile frameworks seek out for a continuous improvement strategy where new developments are early validated and delivered to the user, very much in line with the principles advocated by the DevOps culture or modern testing methodologies such as Test-Driven Development (TDD).

B

European Open Science Cloud

B.1 Roadmap towards the implementation of the European Open Science Cloud

Communication on the European Cloud Initiative: Building a competitive data and knowledge economy in Europe (2016 April 19th)

Although the first references to the term go back to 2015, the *Communication on the European Cloud Initiative* [182], released in April 2016, marked the start of the consultation process in Europe in order to make Open Science a reality through the EOSC, thus responding to the demands of the Big Data phenomenon existing in the scientific research.

In this document, the ***EOSC emerges as an all-inclusive, open virtual environment that will offer cloud-based services to 1.7 million European researchers and 70 million professionals in science and technology in order to manage, store, share and reuse all publicly funded research data***. The EOSC is one of the two legs within the European Cloud Initiative (ECI), underpinned by the high-bandwidth networks and super-computing capacity offered by the European Data Infrastructure (EDI).

The need for an ECI is legitimated by the under-utilization of the full potential of the research data. The EOSC is the response to the existing fragmentation in the European data infrastructures, and shall provide an unified access and better sharing of resources through the federation of the computing and data infrastructures at the national and international level. Box 1.1 summarizes the requirements that need to be met in order to implement the EOSC. The document defines specific actions to promote the fulfillment of those requirements.

Box 1.1: Requirements for realising the EOSC (from [182])

- *Make all the scientific data produced by the Horizon 2020 Programme open by default*, promoting FAIR compliance.
- *Raise awareness and change incentive structures for data sharing*, intended for academics, industry and public sector.
- *Develop specifications for interoperability and data sharing*, across disciplines and infrastructures.
- *Create a fit-for-purpose pan-European governance structure*, to federate data infrastructures, overcoming existing fragmentation in European data infrastructures.
- *Develop cloud-based services for Open Science*, being supported by the EDI.
- *Enlarge the scientific user base* of EOSC to researchers and innovators from all disciplines and Member States.

First report of the High Level Expert Group: Realising the European Open Science Cloud (2016 Oct 11th)

The first report of the HLEG was released as part of a consultation exercise started by the European Commission (EC) to seek expert advice from both the academia and the industry sector. The HLEG report was a definitive push to corroborate the need of an EOSC and urged stakeholders to conduct immediate actions for its implementation [1].

The report highlights the worrying situation of a "clash of cultures" between domain specialists and e-infrastructure specialists in Europe. Both communities, while essential to Open Science, have not closely co-evolved. This situation precludes an agile co-development of core scientific data infrastructures. A side effect of this issue, as the report concludes, is the alarming shortage of data expertise in Europe and points it as pressing requirement for the EOSC realisation. This "core data experts" are required to guide researchers in technical aspects throughout the data discovery cycle.

Box 1.2 compiles the main requirements s pointed out by the first HLEG report.

Box 1.2: Main requirements in EOSC from 1st HLEG report

- *Policy* requirements
  - Affirmative and immediate action on the EOSC in close concert with Member States.
  - Frame the EOSC as the European contribution to the Internet of FAIR Data and Services underpinned with open protocols.
- *Governance* requirements: should be lightweight, internationally effective and define the Rules of Engagement for the service provision.
- *Implementation* requirements:

[1] In particular, the report encourages "immediate action on kick-starting a preliminary phase" and to "close discussions about the 'perceived need' of an EOSC."

- Set the initial guiding principles to kick-start EOSC, endorsing and implementing the Rules of Engagement and provide a clear schedule for the preparatory phase of the EOSC.
- Foster training activities to pursue the development of core data expertise.
- Funding scheme shall be linked to core EOSC areas, rather than being broad and bottom-up topical calls in order to accelerate the development of the EOSC.
- Make data stewardship plans mandatory foll all research proposals.

EOSC Declaration (2017 Oct 16th)

Following up on the consultation strategy, the EC determined the level of commitment of the principal European institutions [183] to the implementation of the EOSC. The resultant EOSC Declaration [80] document, as a result of the EOSC Summit 2017, reflects the institutional support in the initialisation of the EOSC process –sustainable in the long-term, in accordance with the 1st HLEG report– through the articulation of 33 high level statements, summarized in Box 1.3. Those institutions agree on promptly endorse these practices "in their respective capacities".

Implementation Roadmap for the European Open Science Cloud (2018 Mar 14th)

The financial support needed to implement the EOSC vision shaped up in the previous documents was targeted to the European Union (EU) Framework Programme for Research and Innovation, coined as Horizon 2020, in accordance with the 2016 Communication, and in particular, through the Horizon 2020's 2016-2017 and 2018-2020 Work Programmes (WPs). Hence, since 2018 the specific INFRAEOSC call supports the primary objectives of the EOSC in regards to the federation and integration of services, connectivity of pan-European research infrastructures (mainly European Strategy Forum on Research Infras-

tructures (ESFRI)) and the attainment of the complete adoption of the FAIR principles.

Box 1.3: EOSC implementation principles from the EOSC Declaration

- *Data culture and FAIR data*
  - Open-access research data is recognised as a significant output of research, and shall be appropriately managed according to the FAIR principles. Transition to FAIR data needs to be progressive applied through adoption plans coordinated by the EC.
  - The required cultural change needs to be driven by higher education and training, and stimulated through incentives and a reward structure.
  - EOSC realisation requires from trusted (national and European) and FAIR-certified RIs and data repositories. EOSC shall provide researchers with tools to make data FAIR.
  - Data Management Plans (DMPs) are the prerequisite for accurate data stewardship and must be a mandatory part of any publicly-funded research project.
  - Legal barriers for the FAIR implementation shall be addressed by the EOSC. International forums (Research Data Alliance (RDA), Committee on Data for Science and Technology (CODATA)) must be used to reach consensus.
- *Data services and architecture*
  - EOSC envisaged as a one-stop-shop and user-driven data infrastructure commons to serve the needs of scientists, where continuous dialogue between stakeholders –users, funders, providers– guarantee its sustainability.
  - Criteria definition for establishing a prioritized onboarding of resources, components and initiatives, which incentivise reusability of

existing building blocks from past and ongoing projects. Services shall be offered at the highest maturity or TRL.

  - RIs shall improve the utilisation of the EOSC. High Performance Computing –through the EuroHPC initiative– should provide the advanced computing requirements of the EOSC.

- *Governance and funding*

  - A sustainable and interdisciplinary EOSC requires representativity, inclusiveness and transparency, with a 3-layer governance model at the institutional, operational and advisory levels.
  - The funding model shall guarantee the long-term sustainability of the open research data and data infrastructures through the EOSC.

On the other side, the launch of the EuroHPC Joint Undertaking is meant to contribute to the EDI in order to provide the High-Performance Computing (HPC) infrastrucutre to support data computation in the EOSC. In March 2018, the EC sets out the possible –action lines and timelines– implementation of the EOSC, as envisaged by the 2016 Communication, through the delivery of the *Implementation Roadmap for the European Open Science Cloud* document [81] . This document relied upon the outcome of the consultation with scientific stakeholders and builds on the WP 2018-2020 to start implementing the EOSC.

Box 1.4: Six action lines for an EOSC model

- Federated *architecture* that remedies existing fragmentation in research data infrastructures:
  - Building on the existing and successful federated infrastructures, such as EGI Federation or EUDAT Collaborative Data Infrastructure (EUDAT-CDI) (Horizon Work Programme 2016-2017).
  - A first phase for building a *federated core* through the EOSC-hub project that will provide horizontal services such as a portal, authentication and authorisation and security services.

  - A second activity for a progressive federation of a large number of data infrastructures, both EU and national, in terms of resources and services provided thereof. Minimum commitments are set in the Rules of Participation (RoP) [a] only for the specific requirements of the federation. Data infrastructures will remain having entire control of their own rules outside those commitments.

- *Data* stewardship culture through FAIR principles:
  - Progressive development of shared resources and tools used by data-savvy researchers and implemented by the data infrastructures.
  - Consultation to a FAIR data HLEG group, that in 2018 delivered the *Turning FAIR into reality* [23] report.
- Catalogue of interdisciplinary and borderless user-driven *services*:
  - Including the core or horizontal services, adding services for data management and analysis.
  - Initial catalogue will be based on existing services provided by EGI or EUDAT-CDI, and specific thematic services, already being integrated through the ongoing EOSC-hub project.
- *Access & interface* through the EOSC Portal as the universal entry point for all the potential users: EOSC-hub and eInfraCentral projects are piloting a first common platform for accessing to the EOSC shared resources.
- *Rules* of participation for setting out the rights, obligations and accountability of the different stakeholders, such as service providers and users. The EOSCPilot project and the HLEG laid out a preliminary design of such rules that are evolving through the WP 2018-2020.
- *Governance* model based on a representative multi-stakeholder approach that supports the i) long-term sustainability and coordination of the data infrastructure federation, ii) implementation of catalogue of services, FAIR

data, EOSC portal and RoP, iii) definition and monitoring of Key Performance Indicators (KPIs) and iv) reporting.

[a]Previously known as *rules of engagement* in the above-reviewed documents, it was modified according to the military nature of the term

2018 (Nov 21st) - Second report of the High Level Expert Group: Prompting an EOSC in practice

The second HLEG report [79] came later in 2018 to build upon the strategic vision provided by the first EOSC HLEG and having the 2018 Implementation Roadmap "at its very heart".

Box 1.5: New recommendations from 2nd HLEG

- *Implementation*
  - Presence of KPIs in the new INFRAEOSC project workplan
  - Increase the availability and volume of quality & user-friendly scientific information on-line
  - Carry out a landscape analysis on a national level within the Member States
  - Separate RoP in two sets in order to differ between users and providers. For service providers, include a requirement for participation demonstrating sustainability and simplify early (beta) participation by relaxing initial constraints
  - Create a marketplace-like universal entry point to the EOSC to ease access and promote industry reuse of scientific outputs
  - Promote the development of open, sustainable, versioned, documented and energy consumption aware sofware for all elements of the EOSC
  - Set up an EOSC Helpdesk to lower barriers to entry and ensure transparency and support to user engagement

  - Foster EOSC-Public Private Partnership (PPP) to minimise operational risks in completion of the EOSC implementation vision

- *Engagement*
  - Stimulate the 'supply side' to demonstrate the usability and reusability of data and services through RIs
  - Stimulate the 'demand side' to demonstrate the Return on Investment (ROI) via successful in-practice stories

- *Steering*
  - Adopt state-of-the-art technologies that build trust towards a shared security model in EOSC
  - Ensure that the executive board decisions are based on latest scientific and organisational trends

One of the main outcomes is the materialisation of the Minimum Viable Ecosystem (MVE) of the EOSC to represent the smallest workable system that complies with the main EOSC expectations, and the realization of an EOSC governance model to enable the MVE process. The concept of MVE goes in line with the 2018 Implementation Roadmap, which stated that the EOSC would be progressively built in stages. The boundaries of the RoP, built on the FAIR principles and relevant for deliver the EOSC MVE, were defined in two sets according to the audience, separating users and –data, service and infrastructure– providers. The governance model of the MVE emphasizes the 3-layer approach of the 2017 Declaration –strategic, executive and stakeholder– requesting the need of an EOSC board, an Executive board and a Stakeholder forum accountable for each layer. Different working groups have been defined within the Executive Board.

C

European Grid Infrastructure

C.1 Operating system support within the Unified and Cloud Middleware Distributions

Each UMD or CMD major release supported, on average, two Linux operating systems that represented RedHat and Debian-like distributions. Table C.1 shows the evolution of the operating systems being supported throughout the UMD and CMD major releases, which in some cases raised up to 3 different Operating System (OS) distributions.

Major release	Supported OSes
UMD-1	Scientific Linux 5
UMD-2	Scientific Linux 5
	Debian Squeeze
UMD-3	Scientific Linux 6
	Scientific Linux 5 (*)
	Debian Squeeze (*)
UMD-4	Scientific Linux 6
	CentOS 7
CMD-OS	CentOS 7
	Ubuntu 18.04
	Ubuntu 16.04
	Ubuntu 14.04 (*)

Table C.1: OSes supported throughout UMD and CMD distributions lifetime. The support for the OSes marked with an '*' were dropped during the associated release.

Glossary

API Application Programming Interface. 12, 143, 144, 147, 148, 172, 187, 188, 191

CCA Continuous Configuration Automation. 10, 11, 59, 110, 113, 130–132, 162, 163, 165, 166, 174

CD Continuous Delivery. 59, 60, 126, 143, 144

CI Continuous Integration. 10, 59, 60, 121, 122, 124–126, 133, 142, 143, 164, 171, 174

CI/CD Continuous Integration and Delivery. 9, 10, 60, 61, 121–127, 132, 133, 135, 138, 141–143, 174, 179–181, 183, 184, 191, 193, 194, 198, 199

CMD Cloud Middleware Distribution. 10–12, 76, 77, 155–158, 162, 163, 166, 173, 174, 199, 245, 246

CODATA Committee on Data for Science and Technology. 22, 239

CVE Vulnerability and Exposure. 107

DAST Dynamic Application Security Testing. 103, 108, 109, 232

DEEP DEEP Hybrid-DataCloud. 10, 78, 89, 119, 124, 137–143, 147, 149, 151, 154, 171, 180, 183, 186, 191, 193, 198–200

DEEP-OC DEEP Open Catalogue. 10, 143, 145–151

DEEP-OC-app DEEP Open Catalogue application. 143, 146, 147, 149

DEEPaaS DEEP-as-a-Service. 10, 140, 143, 144, 147–151

DMP Data Management Plan. 30, 239

DOAJ Directory of Open Access Journals. 25

EA Early Adopters Programme. 69

EC European Commission. 22, 26, 30, 64, 65, 67, 69, 72, 73, 237–240

ECI European Cloud Initiative. 236

EDI European Data Infrastructure. 236, 240

EGEE Enabling Grids for E-sciencE. 75, 154

EGI European Grid Infrastructure. 10, 65, 74–77, 121, 123, 132, 134, 153–158, 160, 162–168, 171, 172, 174, 175, 200, 228, 229, 240, 241

EGI QC EGI Quality Criteria. 12, 77, 158, 159, 161, 162, 173, 174, 186, 199, 200

EGI SWPP EGI Software Provisioning Process. 10, 11, 77, 154, 156–159, 166–168, 172, 174, 175, 193, 199, 200

EMI European Middleware Initiative. 155

EOSC European Open Science Cloud. 16–18, 23, 64–68, 70–76, 78–80, 83, 84, 145, 180–183, 192, 193, 197–200, 203, 204, 235–243

ESFRI European Strategy Forum on Research Infrastructures. 238

EU European Union. 29, 63, 238

EUDAT-CDI EUDAT Collaborative Data Infrastructure. 65, 74, 240, 241

FaaS Function as a Service. 10, 148, 149, 151

FAIR Findability, Accessibility, Interoperability and Re-usable. 29, 30, 204, 236, 237, 239, 241, 243

FLOSS Free/Libre and Open Source Software. 26, 27, 97, 98, 100

FOSS Free and Open Source Software. 27

GUI Graphical user interface. 230

HLEG High Level Expert Group. 67, 73, 74, 182, 193, 204, 237, 238, 241, 242

HPC High-Performance Computing. 240

HTC High-Throughput Computing. 75, 76

HTML HyperText Markup Language. 145

HTTP Hypertext Transfer Protocol. 124, 147, 172, 173

IaC Infrastructure as Code. 59, 113

ICT Information and Communications Technology. 63

INDIGO INDIGO-DataCloud. 9, 10, 65, 74, 77, 78, 88, 89, 119–128, 130–135, 137, 138, 142, 149, 151, 154, 171, 180, 183, 186, 193, 198–200

ITSM Information Technology Service Management. 70

JOSS Journal of Open Source Software. 101, 102

JSON JavaScript Object Notation. 101

KPI Key Performance Indicator. 242

LERU League of European Research Universities. 30

LTS Long Term Support. 94, 97

ML Machine Learning. 78, 137, 199

MVE Minimum Viable Ecosystem. 74, 243

MVP Minimum Viable Product. 102, 233

NGI National Grid Initiative. 76

OECD Organisation for Economic Co-operation and Development. 22

ORD Open Research Data. 26

OS Operating System. 12, 159, 245, 246

OSPP Open Science Policy Platform. 30, 73

OWASP Open Web Application Security Project. 107, 108

PaC Pipeline as Code. 137, 139, 140, 145, 149

PID Persistent Identifier. 28, 29, 101, 184

PPP Public Private Partnership. 243

PR Pull Request. 98, 99, 103, 108, 112, 122, 124, 131, 145, 171–175

RDA Research Data Alliance. 22, 239

REST Representational State Transfer (REST) is the software architectural style of the World Wide Web. REST was defined by Roy Thomas Fielding [**fielding2000architectural**]. RESTful systems typically, but not always, communicate over HTTP using the HTTP verbs GET, POST, PUT, DELETE, etc. REST systems interface with external systems as web resources identified by URI. 144, 147

RI Research Infrastructure. 63–69, 71, 73, 79, 228, 229

ROI Return on Investment. 60, 243

RoP Rules of Participation. 67, 74, 241–243

RSE Research Software Engineer. 34, 228

RT Request Tracker. 172, 174

SAST Static Application Security Testing. 103, 107, 109, 232

SCM Source Code Manager. 91, 93, 94, 97, 99, 112, 113, 138

SDLC Software Development Life Cycle. 42, 44–46, 50, 51, 53, 54, 57, 59, 61, 154, 156, 166, 183, 198

SE Software Engineering. 42–44, 52–54, 56, 61, 183

SQA Software Quality Assurance. 9, 10, 12, 16, 17, 37, 39, 43–46, 49, 52, 53, 55–58, 61, 71, 75, 77, 78, 83, 84, 88–92, 97, 103, 106, 110, 114, 115, 119–122, 124, 125, 127, 129–135, 137, 138, 141, 142, 149, 154, 156, 159, 179–184, 186–193, 197–200, 204

SQAaaS SQA-as-a-service. 11, 18, 79, 180–187, 190–194, 200, 203, 204

SQAP Software Quality Assurance Plan. 53

SSI Software Sustainability Institute. 227, 228

SYNERGY EOSC-Synergy. 12, 78, 79, 180, 182–184, 186, 192, 193, 200, 204

TDD Test-Driven Development. 129, 233

TOP Transparency and Openness Promotion. 23

TP Technology Provider. 65, 155, 166–169, 171–175

TRL Technology Readiness Level. 69–71, 79, 182, 193, 203, 240

TTM Time to Market. 60

UMD Unified Middleware Distribution. 10–12, 76, 77, 155–158, 163, 165, 170, 173, 174, 199, 245, 246

V&V Verification and Validation. 9, 12, 53–56, 61, 69, 103, 120–122, 124, 134, 151, 187, 189, 229

VCS Version Control System. 91–93, 184, 188

WP Work Programme. 238, 240

XDC eXtreme-DataCloud. 89