

Software Quality Assurance in INDIGO-DataCloud project: a converging evolution of software engineering practices to support European Research e-Infrastructures

P. Orviz Fernandez · M. David · D. C. Duma · E. Ronchieri · J. Gomes · D. Salomoni

Received: date / Accepted: date

This is the author's pre-print version of this work. The DOI of the final publication is 10.1007/s10723-020-09509-z

Abstract From the advent of Grid technology – 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 the scientific community requirements has been addressed in Europe through dedicated software development projects for e-Infrastructure creation, operation and management. This work presents the most significant software quality breakthroughs obtained in one of such projects, *INDIGO-DataCloud*, the main challenges and

P. Orviz Fernandez
CSIC, Santander, Spain
E-mail: orviz@ifca.unican.es

M. David
LIP, Lisbon, Portugal
E-mail: david@lip.pt

D. C. Duma
INFN - CNAF, Bologna, Italy
E-mail: cristina.aiftimiei@cnaif.infn.it

J. Gomes
LIP, Lisbon, Portugal
E-mail: jorge@lip.pt

E. Ronchieri
INFN - CNAF, Bologna, Italy
E-mail: elisabetta.ronchieri@cnaif.infn.it

D. Salomoni
INFN - CNAF, Bologna, Italy
E-mail: davide.salomoni@cnaif.infn.it

barriers confronted throughout the lifespan of the project, and how they were partially or totally overcome. The knowledge base established throughout the last 15 years of diverse software development initiatives in Europe for sustaining distributed research e-Infrastructures, supported by the advances in the area of software engineering, definitely contributed to improve the quality and reliability of the software delivered, and consequently, the operational stability of the European e-Infrastructures. *INDIGO-DataCloud* project is a good evidence of such insights, where, unlike the preceding trend found in past projects, the enforcement of Software Quality Assurance practices has been present since the very early stages of the software lifecycle.

Keywords Software Reliability · Quality Assurance · Software Metrics · Software Testing Techniques · DevOps

1 Introduction

Throughout the last 15 years, the European Commission (EC) has continuously invested on software development initiatives to provide research e-Infrastructures with the capability of supporting unified access to large-scale computing and intense data analysis facilities. The developed tools have enabled the federation of geographically distributed resource providers, with the ultimate aim of supporting the growing needs of the scientific communities. According to their requirements, those research e-Infrastructures have been evolving, exploiting diverse distributed computing technologies namely Grid, in the first place, and, eventually, Cloud.

The urging needs of providing stable e-Infrastructures, which were based on novel distributed computing technologies, initially drove the efforts spent in software development towards the provision of new functionalities, reducing the adoption of software engineering methodologies to a marginal player. The released software was less tested, resulting in a high rate of rollbacks and patches applied to production systems, when compared to the subsequent projects. Thereby, the stability of the e-Infrastructures were often compromised [1], leading to outages in the participant resource centers. Back in the days when Grid middleware was actively developed, researchers reported the need of improvements both in terms of reliability and performance [2], eventually developing their own custom solutions [3] [4] on top of the existing services.

In an effort to improve the operational experience of research communities, it soon became apparent the need for a better balance between the addition of new features and the quality of the released software. Consequently, incoming European funding programmes and open calls progressively stressed the fact of devoting a more substantial part of the software lifecycle to the improvement of the quality and sustainability of the products being delivered. At this point, evolving software engineering practices were considered and gradually adopted in order to define and implement the quality procedures, the organization of cross-functional teams and the deployment of pilot testbeds.

The increasing prominence of software quality procedures observed in the analysis of the last decade of EU-funded projects related to the development of distributed computing solutions –as described in Section 2–, is pictured in the software project herein described: *INDIGO-DataCloud*. As we will thoroughly describe in the next sections, the ground base laid by these preceding projects and the adoption of practices taken from the prevailing software engineering methodologies –in particular the Agile Software Development [5] and the DevOps culture [6]–, were the catalyst to establish a set of Software Quality Assurance (SQA) criteria – as explained in Section 3.1 – that guided the software lifecycle of the software produced.

Following the DevOps principles, the SQA criteria encourages a thorough verification of the source code, inspecting and testing each minor change meant to be included in the production version. Considering that *INDIGO-DataCloud* solution comprises more than 30 software products and 200 source code repositories, such scenario is hardly accomplished without the aid of automation. Consequently, the SQA process builds on the great advancements that the continuous integration (CI) tools have undergone over the last years, as described in Section 3.2. Hence, acting at the early stages of the software development lifecycle brought a fundamental shift when compared with the reviewed *INDIGO-DataCloud*’s preceding projects that dealt with the task of developing software solutions for Grid and Cloud-based e-Infrastructures.

This strategy allows to improve the overall quality of the software, both in terms of sustainability, by checking the compliance with code style standards, and reliability, through the execution of functional and integration tests, ensuring the total absence of regressions. The results presented in this paper show a divergence between the high rate of bug fixes resolved at development time and the significantly lower number of bugs detected by the end users. Consequently, bugs are more effortlessly fixed since they are early captured and end users interface with more reliable and stable software.

The paper is organized as follows. Section 2 details the evolution of SQA processes in the preceding EC-funded projects that developed software solutions for Grid and Cloud-based e-Infrastructures. Section 3 thoroughly describes the new trends in software engineering conducted during the course of the recently finished *INDIGO-DataCloud* project. Conclusions are drawn in Section 4.








2 Background work on European e-Infrastructure projects

One of the fundamental pillars of the *INDIGO-DataCloud*’s SQA process has been established by the outcomes resulted from similar EC-funded e-Infrastructure-related projects over the last 15 years. In particular, in this section we are primarily interested in emphasizing the collaborative software development efforts within the category of projects that improved the accessibility and exploitation of distributed computing technologies in the European research landscape. As such, we will present the most representative software

engineering practices adopted by those projects to understand how they ultimately inspired the SQA process definition and implementation in *INDIGO-DataCloud*, further discussed in Section 3.

The EC projects considered in this paper are listed chronologically in Table 1. Each project paved the way for the following ones, according to a logic evolution in which an increasing awareness in software quality and reliability has been showcased. As shown in Table 2, the most recent projects are more committed to, and aware of, software engineering practices. Here we include *INDIGO-DataCloud* (INDIGO-DC) – see Section 3 – as the reference project where all the listed features have been achieved.

Table 1 List of EC-projects

Logo	Short Name	Long Name	Duration
	DataGrid	Research and Technological Development for an International Data Grid	2001–2003
	EGEE I, II, III	Enabling Grids for E-science	2004–2010
	ETICS 1, 2	E-Infrastructure for Testing, Integration and Configuration of Software	2006–2010
	EMI	European Middleware Initiative	2010–2013
	EGI-Inspire	Integrated Sustainable Pan-European Infrastructure for Researchers in Europe	2010–2014
	EGI Engage	Engaging the EGI Community towards an Open Science Commons	2015–2017
	INDIGO-DataCloud	INtegrating Distributed data Infrastructures for Global ExpLOitation	2015–2017

2.1 DataGrid

The *DataGrid* [7] project (Jan 2001 – Dec 2003) main goal was to provide scientific communities (including physics, biology and earth sciences) with intensive computing and large-scale dataset analysis capabilities. The project brought together 21 academic and industry partners, from 15 different countries [9]. Grid was the emerging technology to be used in order to address their requirements, thus the project delivered its own software distribution, named EDG (EU *DataGrid*), strongly based on Globus middleware components and services [10]. The project was organized in 12 work packages, from which 5 were devoted to software development and coordinated by an Architecture

Table 2 Features on software reliability in EC-projects

Feature		DataGrid	EGEEs	ETICSs	EMI	EGIs	INDIGO-DC
Architecture Task Force		✓	✓	✓	✓	✓	✓
Communication Handling		✓	✓	✓	✓	✓	✓
Requirements Handling		✓	✓	✓	✓	✓	✓
Agile Methodologies			✓	✓	✓	✓	✓
Source code inspection					✓		✓
Build/Testing Management Procedure		✓	✓	✓	✓		✓
Software Product Metrics		✓	✓	✓	✓	✓	✓
Quality Criteria Definition		✓		✓	✓	✓	✓
Automatic Certification				✓		✓	✓
Auto-generated Documentation			✓	✓	✓	✓	✓
DevOps practices adoption (CI, CD)							✓

Task Force, supervising the overall design and technical consistency of the developments.

Without previous experience in such widely collaborative projects, a big effort was spent in devising solutions for several challenges [11], namely 1) the communication overhead (as a result of the large geographical separation of the involved parties in the development tasks), 2) the fast evolution of the requirements from the user communities (50 use cases from the three scientific area), and 3) the lack of a body of knowledge for academic software engineering.

The Agile manifesto [5] was by then an emerging methodology, not yet considered by the project. As such, the project’s development design suffered from the lack of the methods, tools, techniques and best practices emerging from this new discipline of software engineering [12]. Nevertheless, the project focused on the experiences and procedures used by diverse collaborative open source software projects, such as Linux and Apache, trying to achieve a higher maturity level [13].

Project monitoring and reporting mechanisms allowed to assess the risks, efforts and documentation produced. Development guidelines [14] were produced, covering the various phases, including 1) packaging, 2) test and validation, and 3) style and naming convention. A set of quality indicators [15] were defined to measure the efficiency of the system from the user perspective.

Due to the developmental nature of DataGrid project, they solely consider the *crude efficiency* metric as the relevant indicator to track the quality and performance of the core software, also known as *middleware* in Grid terminology. This metric considered the success rate of all the jobs submitted by users using the production testbed, so that all the failures of the middleware were included in the statistic [8]. However, it also accounted for the failures in the remaining pieces in the job submission chain, such as the user application software or the targeted datacenter deployment. Even so, as it can be observed in Fig. 1, the crude efficiency showed a positive trend over the months that followed, with the unique exception of the LCG-1 middleware distribution.

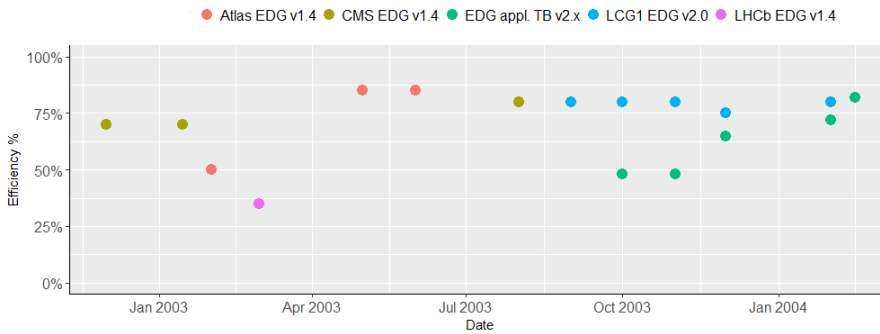


Fig. 1 Efficiency % = Number of jobs successfully completed / Total Number of jobs submitted: results are provided for the different DataGrid's EDG middleware distributions released throughout the project. Note that the figure includes metrics of the same release (EDGv1.4) for three Large Hadron Collider (LHC) experiments.

2.2 Enabling Grids for E-science

The three phases of Enabling Grids for E-science (*EGEE*, Apr 2004 – Apr 2010) [16–18] projects brought together scientists and engineers from over 240 institutions in 45 countries aiming to provide a seamless Grid infrastructure for e-Science. *EGEE-II* and *EGEE-III* featured the internationalization (outside Europe) of the project, embracing worldwide research institutions and user communities [19]. The software to sustain the increasing requirements coming from the diverse scientific communities needed developing a rich set of new services, while maintaining a sustainable infrastructure for Grid computing. This e-Infrastructure was eventually used by more than 15 thousand researchers and deployed in over 250 institutions [20].

The *gLite* middleware [21] was the official software distribution of *EGEE* as of 2006, after two years of prototyping and re-engineering efforts to converge with the *LHC* Computing Grid (LCG-2), Virtual Data Toolkit (VDT) and Condor [22] software stacks. The development team was comprised of

more than 80 people from 12 academic and industrial partners, which issued more than 10 thousand bug fixes, 1.7 thousand patches and defined over 300 development tasks tracked using bug/task management tools.

The source code was available at a private, centralized version control system. The code passed through a manual certification procedure at the time of the release. This procedure aimed to improve the reliability of the software components by applying acceptance criteria checks at the pre-release stage [23] i.e. integration, certification, pre-production and production. Starting with *EGEE-II*, the project adopted automation in the software lifecycle process by leveraging the automatic build system for Grid middleware, that is the *ETICS* [24] solution.

2.3 E-Infrastructure for Testing, Integration and Configuration of Software

The E-Infrastructure for Testing, Integration and Configuration of Software [24] (*ETICS*, Jan 2006 – Feb 2010) project aimed at addressing the challenges in producing quality software in distributed, collaborative projects such as *EGEE* and its *gLite* middleware. The *ETICS* framework integrated different technologies and tools in order to provide automated configuration, build and testing capabilities, as well as auto-generated documentation and software metrics gathering – such as Source Lines Of Code (SLOC), complexity and number of defects/bugs [24] –. The *ETICS* framework was the first automated service for delivering quality software products in distributed environments like the Grids.

2.4 European Middleware Initiative

The European Middleware Initiative (*EMI*, May 2010 – Apr 2013) [25] project joined the 4 major Grid middleware providers in Europe at the time – *gLite*, *UNICORE*, *ARC* and *dCache* – with the goal to maintain and evolve the middleware focusing on extending their interoperability and improving the reliability of the services. The ISO/IEC 9126 [26] standard was used in order to identify a set of characteristics that needed to be present in the *EMI* software products and processes to be able to meet the *EMI* quality requirements [27].

For each software characteristic, a set of associated metrics and Key Performance Indicators (KPIs) were identified and defined in detail in the *EMI* Metrics Specification [27]. The project leveraged the *ETICS* service for the development, continuous integration and release management, as well as for metric tracking, making queries on the collected data to display them through a chart generation framework.

2.5 EGI–Integrated Sustainable Pan–European Infrastructure

The *EGI–InSPIRE* (Integrated Sustainable Pan–European Infrastructure for Researchers in Europe, May 2010 – May 2014) project [28] was the continuation of the *EGEE–III* project, with the objective of establishing and maintaining a sustainable European Grid Infrastructure, composed by a federation of National Grid Initiatives and interoperable with other Grids worldwide. This goal would be accomplished through the development and maintenance of various operational tools – such as the Operations Portal [29], the *EGI* Helpdesk [30], or the Grid configuration database (GOCDB) [31] – and the management of the software provisioning process (SWPP) [32], dealing with the validation and distribution of the software to the production infrastructures. As a result, this process led to a production-ready Grid computing middleware distribution named *UMD* (Unified Middleware Distribution). At that time the *UMD* was a stack of about 250 software components from several technology providers, such as *EMI* and *Globus*. In order to be distributed through UMD repositories, the software had to be compliant with EGI’s Quality Criteria (QC) [33]. The QC defined a set of requirements in different areas: documentation, deployment, security, information model and operations.

The impact of the quality assurance activities was observed once the *EMI* software contributed to the *EGI–InSPIRE* project. The data reported in Table 3 (see page 13 in [27]) summarizes the *EMI* software quality evolution per project quarters (PQ) as evaluated by the EGI project with the help of the *UMD* QC. We can observe the improvement in quality of the *EMI* releases over time, measured by the number of products that both met the software distribution criteria, by means of the *UMD* QC definition, and the EGI Stage Rollout (SR) phase. This latter phase completed the criteria validation by deploying the *UMD* QC–certified software in a set of candidate production sites.

Table 3 *EMI* software quality evolution

PQ	Release (RP)	Products	N. RP UMD QC	Passed	N. RP SR	Passed	N. RP UMD QC	Failed
5	30		27		27		0	
6	30		28		26		2	
7	27		26		24		2	
8	18		18		18		0	

UMD is still being used and deployed in the European scientific e–Infrastructures under the follow–up project *EGI–Engage* (Engaging the EGI Community towards an Open Science Commons) [34]. This distribution is currently complemented by a Cloud–specific one called *CMD* (Cloud Middleware Distribution). The increase in the number of products that the new *CMD* distribution brought in, compromised the effectiveness of the SWPP realization, in particu-

lar the software validation phase. The modernization of the EGI QC validation was accomplished through the adoption of automation, by means of the programmatic evaluation of its fundamental quality requirements [35].

3 INDIGO–DataCloud Software Quality Assurance Process

INDIGO–DataCloud project [36] leverages the lessons learned from the previous experiences described in Section 2. The expertise gathered throughout these years was highly profitable regarding the adoption of state-of-the-art software engineering methodologies, the management of new technologies to put those methodologies into practice and the analysis of an appropriate set of metrics to measure the quality of the implemented solutions. All of which with the additional complexity of being framed under collaborative environments with an extensive list of partners.

However, new challenges appear in the course of a software development project, which in some cases are tackled using novel software engineering approaches. In this regard, *INDIGO–DataCloud* faced analogous challenges as the ones reported by parallel EC-funded software development initiatives [37] that were addressed using similar solutions, such as relying on DevOps practices for tackling the software lifecycle management. The SQA process followed by *INDIGO–DataCloud* combines these solutions with pragmatic approaches built upon external and first-hand experiences. In the following sections, we describe the three main pillars that sustain the project’s strategy for attaining quality in the software produced:

1. The definition of a *Software Quality Assurance (SQA) criteria and a metrics gathering procedure* to provide guidelines aiming at developing quality software, validating each change in the code, to facilitate its adoption. The software lifecycle is continuously monitored by a set of metrics, allowing prompt reactions to detected malfunctions that may occur throughout the process.
2. The *promotion of automation* to enable the foregoing change-based validation approach and to accelerate the delivery process of new software versions.
3. A post-release validation to be carried out over the project’s *preview testbeds*, where the new product versions are deployed as part of the integration validation, and the *testing in production environments* by external resource providers.

As we will describe hereafter, the *INDIGO–DataCloud* SQA process is not restricted to software targeted to distributed computing environments, but instead oriented to the management of any form of collaboration driven software development, composed by multiple software projects that collaborate jointly. However, there are particularities, such as the stage-rollout phase, that are attached to an e-Infrastructure’s operation, thus might not be applicable to all cases.

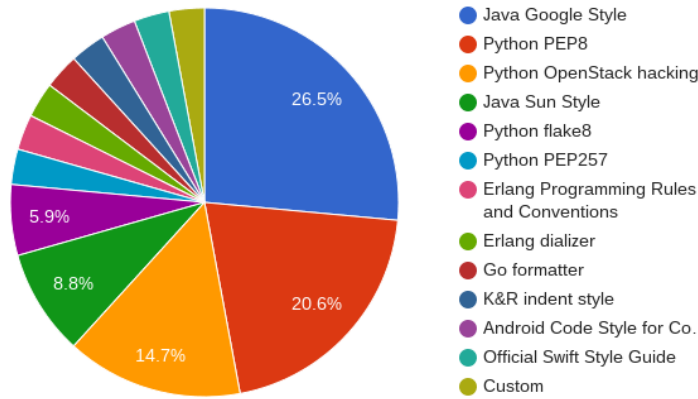


Fig. 2 Code style standards followed by *INDIGO-DataCloud*'s software products.

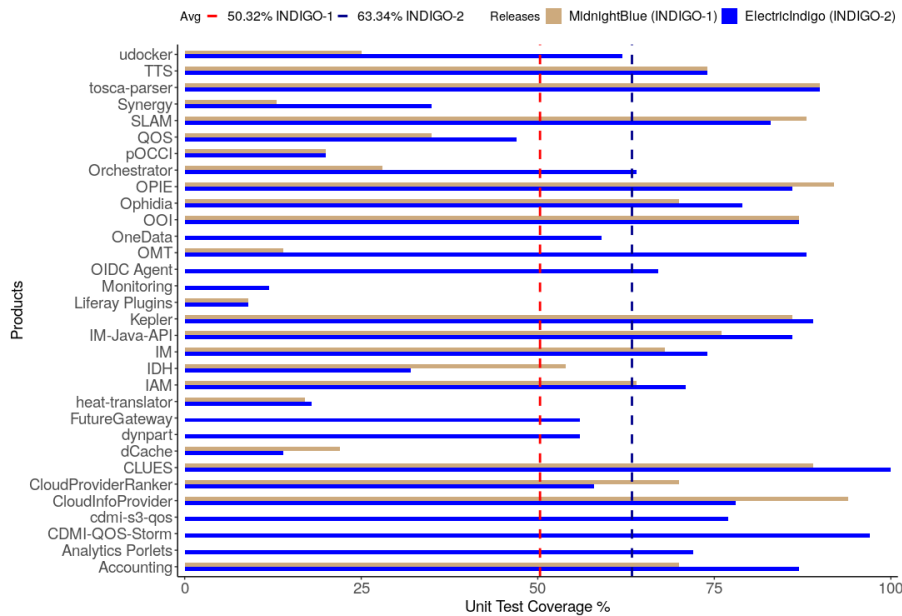


Fig. 3 Comparison of the unit testing coverage values for the *INDIGO-DataCloud* software stack over the *INDIGO-1* and *INDIGO-2* major releases. Products that were exclusively released during *INDIGO-2* do not show data for the first period of *INDIGO-1*. Source code for all the *INDIGO-DataCloud* components can be found in the GitHub's *indigo-dc* organization [52]

3.1 Software Quality Procedures

The software quality policies were initially defined in the first deliverable document of the project [38]. The policies were reviewed periodically taking into account the requirements from the user communities, the feedback from the

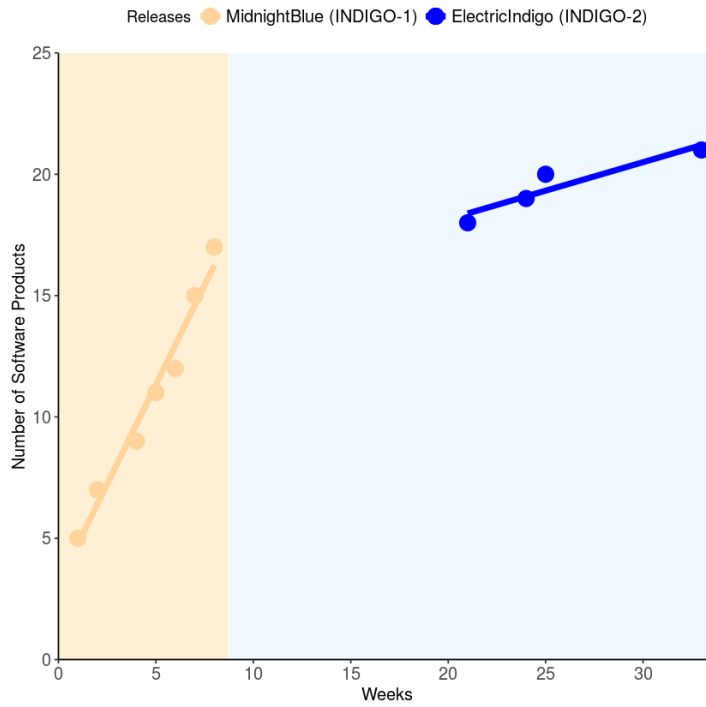


Fig. 4 Adoption of Configuration Management tools throughout the project lifetime. The figure shows the trend lines leading to the first (light cream points and line) and second release (dark blue points and line).

development teams and the insights of software engineering practices. These policies covered 1) the identification and description of the *SQA requirements* that the produced software needs to comply with, and 2) the *quality metrics*, to monitor each software product's behaviour throughout the development, release and post-release stages.

3.1.1 SQA criteria

The SQA criteria are a set of conventions and recommendations that pave the way for the adequate development, timely delivery and reliable operation of the produced software components. They emphasize the quality requirements and best practices to be applied at the development phase, such as style and testing compliance or human code reviews, to protect the production source code versions. Furthermore, the criteria promote the adoption of the software by providing minimum requirements of documentation content and stressing the usage of automated solutions for software deployment. The SQA criteria are publicly available [39] and it is continuously evolving by means of an open collaboration procedure [40] that aims at serving as a reference for quality assurance realization in research software.

- *Code style.* The ultimate goal for code style assessment is to improve the readability and reusability of the source code produced under the scope of the project. Code style standards are enforced for every software component in the project stack. Community most-adopted or *de-facto standards* are initially recommended, however the development teams eventually choose the set of guidelines to comply with. Fig. 2 highlights the code style standards and their popularity in the *INDIGO-DataCloud* software stack.
- *Unit and Functional testing.* Changes involving the addition of new functionalities are required to be tested. *Regression testing* in this context is accomplished by enforcing the definition and periodic execution of the tests, covering past fixed issues, to ensure that they are not reintroduced during the development activities. *Unit testing* completes the source code testing evaluation by focusing on the code’s internal design. The SQA criteria of *INDIGO-DataCloud* set a recommended threshold of 70% coverage for unit testing. In Figure. 3, unit testing coverage is compared for each software component between the two major releases, denoting an incremental trend over the course of the project: code coverage increased from an average value of 50.32% to 63.34%, outlined by the respective dates of the project’s first (*INDIGO-1*) and second (*INDIGO-2*) releases. Nevertheless, Fig. 3 shows a decrease in the unit testing coverage values for 18% of products. As unit tests cover low-level code elements, they are best considered while the new code is being written [41]. We have observed that this decrease was aligned with high-demanding periods of software development, as seen in the previous months of the *INDIGO-2* major release. Regardless of the potentially higher defect density that this fact brought along, the overall performance kept progressing as the rest of the software stack substantially improved their own unit testing coverage statistics. By *INDIGO-2*, 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. The individual records followed also a growing trend (only 6 out of 32 components had lower values in *INDIGO-2*). It is worth noting that Fig. 3 not only includes new software developments implemented from scratch within the project. Software tools and libraries from external open-source projects – such as *tosca-parser* or *heat-translator* –, contributed upstream by the project, and well-established products, involved in *INDIGO-DataCloud* but not contributing with the 100% of their codebases – such as *dCache* –, are also considered in the analysis. However the values given apply to the entire codebase, even for the latter case of products. These products challenged the application of the SQA policies described in this section, by two means:
 - Previously developed code was not refactored since it was not *owned* by the project.
 - An agreement on prevailing SQA policies for such cases where quality practices were already in place.

- *Integration testing.* Software components usually interact with other services during operation. Integration testing deals with the interactions among coupled software components or parts of a system that cooperate to achieve a given functionality. This type of testing might be complex and, based on the project’s experience, difficult to be implemented in an automatic way. The aim is to guarantee the overall operation of the component with regard to the services it interfaces with, whenever new functionalities are introduced.
- *Code review.* This phase is the last step in the change management pipeline, once the candidate change has successfully passed through the testing methods described previously. It implies the human-based revision of the proposed change to discuss its adequacy in terms of e.g. scope, objective fulfilment, and documentation completeness. On approval, the candidate change is definitely merged into the source code’s production version. Human code reviewing is paramount in the software quality assessment, specially important when the SQA testing requirements are fully automated. Secure code reviews are also done at this stage, assessing common vulnerabilities from inputs coming from automated linters and manual dynamic application security testing.
- *Documentation.* The SQA criteria set the path for the adoption of the developed software by defining the documentation content, according to the target audience. This requirement also promotes the automation, both in terms of documentation creation and service deployment. The *documentation is treated as code*, using a markup language, automatically rendered and uploaded to online repositories [42]. Thus documentation is portable and human-reviewed, using the same workflow as the code does, validating the changes before being updated into the production repository.
- *Automated Deployment.* To lower the barriers of software adoption, the SQA criteria require the automated deployment of the products delivered as part of the catalogue. Automation in this context is tackled using configuration management tools. These tools allow the adoption of *Infrastructure as Code* (IaC) practice, managing the component’s deployment through declarative definitions. The definition files appear as an additional source of documentation – self-documenting code – as they sequentially guide the component’s deployment process on multiple platforms. *INDIGO-DataCloud* project contributed to open-source IaC tools such as Ansible and Puppet [43]. A representative example of such contributions are the *50 roles* developed from scratch and currently hosted in the Ansible Galaxy portal [44]. Fig. 4 shows the number of products that offer an automated means for deployment. It shows an increase in the adoption of such tools leading to the *INDIGO-1* release (light cream points and trend line), as well as to the *INDIGO-2* release (dark blue points and trend line). The rate of adoption 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.

3.1.2 Quality metrics

The evaluation of the software quality is performed by measuring the values of the metrics and Key Performance Indicators (KPIs) defined based upon the ISO/IEC 9126 standard. These metrics cover the development, release and maintenance phases of the software lifecycle.

Development metrics are obtained programmatically from several sources, namely GitHub API [45] and the Jenkins [46] service, and graphically displayed as GitHub pages using the GrimoireLab framework [47]. Per-component weekly reports, including the SQA requirement fulfilment, are issued and individually discussed through the different communication channels with the development teams. Issue tracking metrics and KPIs are an essential piece of information of both feature addition and defect solving, useful to detect and fix misbehaviours while in the development phase.

Release metric sources are the online repository servers, namely the Linux package [48] and DockerHub [49] repositories. Jenkins server also contains valuable release data as it is the service where the packages and containers are being built before being uploaded to the online repositories. Release KPIs primarily focus on the frequency and efficiency – mainly rollbacks – statistics.

Maintenance and user support metrics are key for continuously improving the response to issues reported by external users. Feedback is collected from outside helpdesks, such as EGI's GGUS [30], and the GitHub Issues tracker.

3.2 Software quality validation and delivery automation

Tackling the granularity of the former SQA requirements – most of which accounted in a per-change basis – is hardly achieved without the aid of automation. The introduction of automation to validate the quality requirements increases the overall reliability of the produced software: automated testing is more time-efficient, leading to higher code coverages and increased defect detection, when compared with a manual approach [51]. To put it in numbers, as Fig. 5 showcases, the total number of defects detected in the pre-production stage, i.e. throughout the integration testing phase, surpassed by a factor of 30 the ones reported by external users, over the lifetime of the *INDIGO-DataCloud* project. An accurate estimation of the software bugs detected – and fixed – with the aid of the CI/CD infrastructure, previously conducted at the source code analysis stage, cannot be given as those bugs are usually fixed on the fly without being tracked. Nevertheless, we can safely state that they exceed by far the total amount of 200 bugs that the Fig. 5 shows.

As any new change, involving a new feature or fix, passes through the automated SQA machinery, the software is always ready to be released. As a result, the often compromised harmonization between the continuous release of new features – pushed by the development teams – and the maintenance of stable production systems – demanded by users and resource providers – can be guaranteed. The DevOps culture theorizes and provides practical solutions

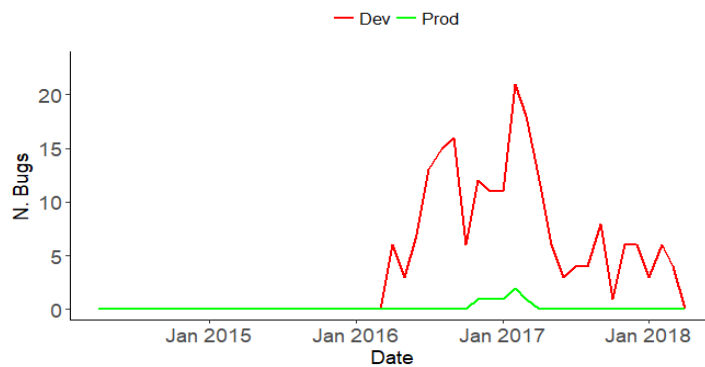


Fig. 5 Number of software bugs – documentation typos are excluded – detected in development and production stages over the lifetime of the *INDIGO-DataCloud* project. Production defects correspond to software bugs filed by users of the EGI e-Infrastructure throughout the next 2 years after the *INDIGO-1* major release, in August 2016 (source: EGI’s GGUS helpdesk). The development defects stem only from the pre-release testing phase, filed exclusively as part of the SQA team work (source: GitHub Issues tracker). As a consequence, the development defects started to arise in the previous months prior to the *INDIGO-1* release, while the major activity was concentrated in the months that preceded the second major release of *INDIGO-2*, in April 2017. As it can be observed, the offset between the advent of defects in development and production corresponds to nearly half a year, with the first production defect reported 3 months after the *INDIGO-1* release.

that aim at unifying both software development (dev) and software operations (ops) teams, by emphasizing the SQA techniques to avoid infrastructure disruption whenever new developments are deployed into production systems. Starting with a Continuous Integration (CI) scenario at the first stages of the project, the software delivered was eventually validated and distributed using Continuous Delivery (CD) pipelines.

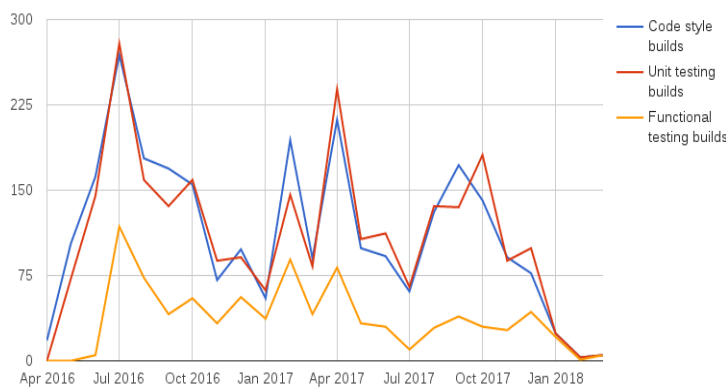


Fig. 6 Evolution of the total number of testing builds triggered automatically as part of the Jenkins CI implementation.

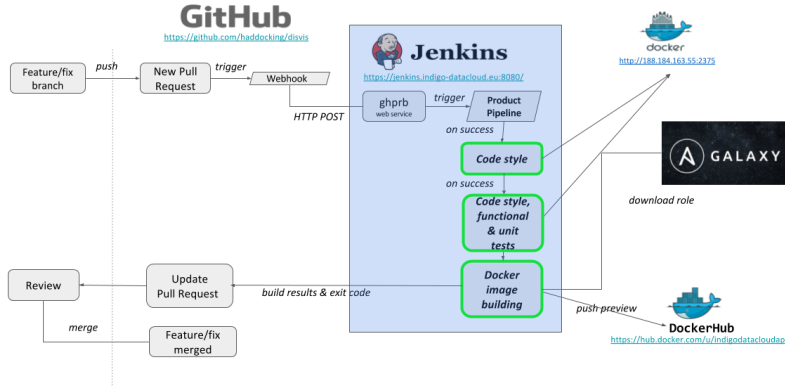


Fig. 7 Continuous Delivery workflow for Docker images.

3.2.1 Continuous Integration (CI)

The *INDIGO-DataCloud* project promoted the application of a CI scenario that enforced, for any piece of produced software, the testing requirements defined in the SQA criteria, as described in Section 3.1. Such an environment requires an automation ready-to-go infrastructure where the different services and technologies involved interact with each other to trigger the source code validation pipeline for each candidate change. The pipeline is mainly comprised by the code style validation, unit and functional testing coverage. The CI pipeline is complemented with additional quality checks, such as integration tests for the software components where automation of this type of testing is applicable, or security linters for the static code analysis of suspicious constructs that could lead to security risks. Metrics gathering is also a part of the CI pipeline to have a per-change trend evolution of the common source code related metrics, such as SLOC or the cyclomatic complexity.

The practical implementation of the CI scenario leverage from tightly integrated open source tools such as:

- GitHub [52] as the online source code repository hosting service,
- Jenkins as the event-response CI,
- Docker container provisioning system, to provide instant computing power needed for executing the pipeline jobs.

The project defined a source code contribution workflow, based on GitHub Pull Requests (PRs), where each change automatically triggers in Jenkins the associated quality and metric tracking checks on PR creation or update.

The CI pipeline is composed from a set of required checks, their exit status is reported back to GitHub. The changes can be prevented if those tests are

not successfully executed. As each change is 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 in this phase.

Fig. 6 shows the evolution, throughout the *INDIGO-DataCloud*'s first and second releases, of the required test types builds since the implementation of the CI infrastructure. Automated functional testing coverage were not available for all the software stack, thus the associated number of builds are fewer. Towards the end of the project, the software development activity slowed the pace but not completely ceased. The support of the CI infrastructure, once reached the project's end of life, allowed the continued maintenance and development of the prevailing software. Some development teams deployed parallel CI systems, taking advantage of the experience gained during the project.

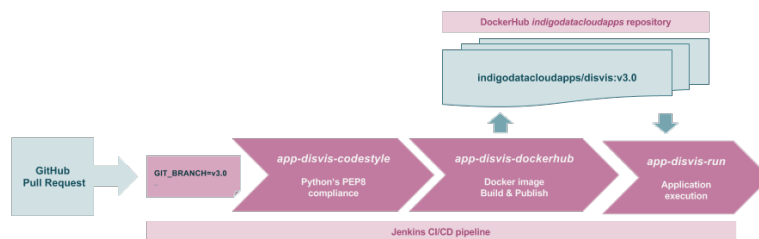


Fig. 8 DevOps pipeline to distribute Docker images for Disvis application.

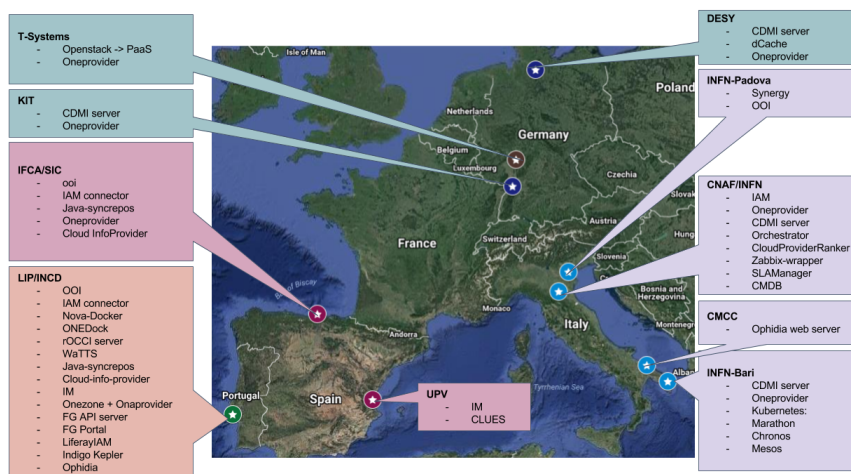


Fig. 9 Resource centers supporting the Pilot Preview testbed and corresponding set of deployed *INDIGO-DataCloud* components or services.

3.2.2 Continuous Delivery

As DevOps suggests, frequent releases positively affect the reliability of the software as they reduce the time in which a given defect is exposed or has an impact, allowing the development teams to act promptly based on the regular feedback [53]. The software updates of *INDIGO-DataCloud* products, which have been taking place since the second major release, are passing through a Continuous Delivery (CD) pipeline that adds the packaging of the software right after the successful execution of the CI pipeline previously described. Consequently, CI-validated software components are automatically delivered in online repositories. Nonetheless, expert supervision to validate the results is required.

INDIGO-DataCloud software stack is delivered in the form of Linux software packages (rpms, debs) and Docker containers. The CD approach is different for each type of software packaging. *Linux package pipeline* intentionally delivers the packages produced in a pre-production – preview – branch. Before that, once the software packages have been created, they are uploaded to a testing branch. The available automated deployment solution – see Section 3.1 – checks the installation and configuration of the component relying on the testing repository. On successful completion, the packages are automatically signed and subsequently moved to the preview branch. In the last step, the release manager supervises the process to move the packages to the production branch.

The *Docker container pipeline* uses the automated deployment solution to actually install and configure the software component in the container image. Once this is done the recently created image is uploaded to the production DockerHub repository, tagged with the label corresponding to the current *INDIGO-DataCloud* release. Fig. 7 shows the complete workflow of the Docker container CD implementation, automatically triggered by a change in the source code.

3.2.3 DevOps adoption from user communities

The experience gathered throughout the project regarding the adoption of different DevOps practices is not only useful and suitable for the software related to the core services in the *INDIGO-DataCloud* solution, but also applicable to the development and distribution of the applications coming from the user communities.

Two applications coming from the supported research communities, DisVis [54] and PowerFit [55], were integrated into a similar CI/CD pipeline described in section 3.2. Fig. 8 shows the pipeline for the DisVis application.

User application developers were provided with both a means to validate the source code before merging and the creation of a new versioned Docker image, automatically available in the *INDIGO-DataCloud*'s application repository.

The novelty introduced in the pipeline above is the validation of the application. Once the application is packaged as a Docker image, and subsequently uploaded to the DockerHub repository, it is instantiated in a new container to be validated. The application is then executed and the results compared with a set of reference outputs. This pipeline implementation goes a step forward by testing the application execution for the latest available Docker image in the catalogue.

3.3 Integration, preview and early adoption

Two pilot infrastructures, integration and preview, were at the disposal of developers and use cases, respectively, involved in the project. The aim of these testbeds is to test the level of integration between the core *INDIGO-DataCloud*'s components and the user applications.

The integration testbed was primarily targeted and used by the software developers. There, unstable software releases were tested by the maintainers themselves in order to validate the interactions with coupled services. Computing and storage resources were occasionally needed, meaning that no real testbed – with all the *INDIGO-DataCloud*'s components – was needed.

Conversely, the preview testbed was actively maintained, where the last stable version of the software was deployed by the SQA team. On top of that, the user applications were deployed in order to test the integration with the *INDIGO-DataCloud* solution. The preview testbed proved to be an effective tool to uncover bugs by means of testing the basic functionalities of the core components. The bugs tagged as *Dev* in Fig. 5 were spotted this stage. A map of the Pilot Preview infrastructure is depicted in Fig. 9. It shows the resource providers and the services deployed. The number of resources steadily increased as the project progressed, due mainly to the progressive increase of supported services and the occasional need for deploying more than one stable version of the same software component in different providers.

Computing and storage resource consumption, required by the pilot testbeds and SQA services (metrics, CI/CD infrastructure), was never an issue for the project's resource providers. In the case of computing resources, the massive use of container-based virtualization alleviated the need of resources as it allowed CPU and memory over-subscription, when no performance test was carried out. Most of the providers were already members of the EGI FedCloud, providing medium-sized Infrastructure-as-a-Service Cloud frameworks with enough computing and storage capacity to meet the deployment requirements of more than 30 products that comprised the *INDIGO-DataCloud* solution and the related SQA services. Nevertheless, the pilot infrastructure coordination task distributed the deployments according each provider's capacity and effort received within the project. Consequently, Fig. 9 shows an unbalanced distribution of services.

As the final validation step, the released software was tested in production environments through the *staged-rollout* process, before its inclusion in the

EGI’s Cloud Middleware Distribution (CMD). This process selected resource providers from the EGI FedCloud infrastructure in order to install the last stable versions of the *INDIGO-DataCloud* software, giving user access to them. The staged-rollout process is key to detect and mitigate issues that could only appear in production environments.

4 Conclusion

Both reliability and sustainability of the *INDIGO-DataCloud* software products have been primarily improved by acting at the source code level. According to the size of the project, which comprised more than 30 products and 200 GitHub repositories, this strategy required a remarkable preliminary ground work. On the one hand, the definition of a set of SQA requirements – compiled in the described SQA criteria – that ought to be fulfilled by the set of core products before being released. On the other hand, the setup of a testing infrastructure and the associated CI/CD pipelines to guarantee that the changes done at the source code level match those requirements in an automated fashion. As a final step, the validation of the resultant pre-released software versions, firstly tackled through integration testing in the project’s preview testbed, and subsequently complemented by the EGI’s staged-rollout process, where the brand new versions are adopted by candidate resource providers within the e-Infrastructure – accessible for user evaluation –, before being tagged as production-ready software, and consequently, distributed through the official EGI channels.

The most noticeable metric that demonstrates a reliability improvement is represented by the ratio between the number of software defects uncovered throughout the pre-release phase of each of the major releases, *INDIGO-1* and *INDIGO-2*, and the ones reported by the end users within the EGI e-Infrastructure, where the core products were deployed. The number of bugs detected throughout the SQA process here described surpassed by a factor of 30 the number of bugs reported by those end users, over a total of almost 200 bugs in the observed timeframe. Considering that this low rate of bugs – taking into account the amount of software products existing in the project – correspond to the outcomes of the integration testing phase, we infer that the preceding phase of code analysis – comprised by the style standard compliance, as well as the unit and functional test cases execution for each minor change – had a substantial impact on the robustness of the resultant software.

Nevertheless, the individual products of *INDIGO-DataCloud* did not undergo a progressive quality improvement throughout its 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 (*INDIGO-2*). This fact demonstrates that, even considered as best practice, test cases are not commonly written right after, or even before, introducing a new unit or functionality in the code. High coverage values

are generally advisable but they entail time-consuming work that is often neglected by the code owners. Rather than requiring a high coverage value, our experience showed that increasing the coverage of those sections in the code that are more likely to cause negative effects to end users is not only a more effective approach to improve software reliability, but it also keeps developers motivated in the tedious task of test coding, as it brings more noticeable benefits for the end user.

All along the project we have observed that source code review is the cornerstone to improve the effectiveness of the unit and functional – when applicable – test cases. *INDIGO-DataCloud* source code repositories are protected against direct pushes so that the code review stage always takes place. Code reviewers analyse the content of each change in the source code and the suitability – according to the SQA criteria – of the associated tests, being the last checkpoint before the change is added into the production branch. Hence the vital importance of providing the tests alongside the code for every given change. However, by the time of *INDIGO-1* only 30% of the total of products provided automated functional tests, reaching almost 60% in *INDIGO-2*. Although test reports were asked in those cases where automated functional tests were not provided, 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 some cases, the unawareness about their existence. We infer from this experience that automated functional tests should be a goal in modern software development as they complement the structural quality checks by adding the validation of the software’s identified functionalities, key in the operational impact and user acceptance.

The application of the formerly described SQA process was not only beneficial for the core products’ quality and reliability, but it has proven to be equally effective for those applications uniquely supported by research communities with little background on software engineering. This fact was demonstrated by the elaboration of an ad-hoc validation pipeline for one of the use cases supported in the project. The CI/CD pipeline extended the source code testing and delivery – already existing in the pipelines for the core products – with a basic system-like testing. For the latter, the pre-packaged application was tested with a pre-defined set of inputs and expected outputs. Only if successfully validated, the source code could be merged into production and the resultant pre-packaged application would be delivered. When this pipeline was integrated in the usual workflow of the application development, it was reported to have constituted a real breakthrough, allowing a faster development and more reliable software, as it was programmatically tested before being distributed. Consequently, the benefits of a continuous SQA improvement process are not only suitable for experienced software developers, but also to any type

of software practitioner, such as the numerous ones commonly found within the research communities.

The *INDIGO-DataCloud*'s SQA process has persisted once the project concluded, being extensively used by the supported software products. The SQA criteria is actively maintained to include new recommendations and best practices, not only targeted for experts but for any enthusiast on research software. Hence, they have established the foundations for knowledge transfer capabilities, which we foresee it will be convenient for the rapid adoption of SQA practices in the subsequent software development initiatives and projects for e-Infrastructure building and management, taking a step forward in the overarching goal of delivering reliable software in research.

Acknowledgment

DataGrid - Research and Technological Development for an International Data Grid project has received funding from the European Union's Fifth Framework Programme under grant agreement IST-2000-25182E.

EGEE - Enabling grids For E-science project has received funding from the European Union's Sixth Framework Programme under grant agreement INFSO-RI-508833.

EGEE - Enabling grids For E-science-II project has received funding from the European Union's Sixth Framework Programme under grant agreement INFSO-RI-031688.

EGEE - Enabling grids For E-science-III project has received funding from the European Union's Seventh Framework Programme under grant agreement INFSO-RI-222667.

ETICS - E-Infrastructure for Testing, Integration and Configuration of Software project has received funding from the European Union's Sixth Framework Programme under grant agreement INFSO-RI-026753.

ETICS - E-Infrastructure for Testing, Integration and Configuration of Software - Phase 2 project has received funding from the European Union's Seventh Framework Programme under grant agreement INFSO-RI-223782.

EGI-InSPIRE - European Grid Initiative: Integrated Sustainable Pan-European Infrastructure for Researchers in Europe project has received funding from the European Union's Seventh Framework Programme under grant agreement INFSO-RI-261323.

EGI-Engage project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement RIA-654142.

INDIGO-DataCloud project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement RIA-653549.

References

1. D. Lingrand, J. Montagnat, J. Martyniak and D. Colling, "Analyzing the EGEE production grid workload: application to jobs submission optimization," *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 37–58, May. 2009.
2. S. Campana *et al.*, "Analysis of the ATLAS Rome production experience on the LHC computing grid," *IEEE 1st Int. Conf. of e-Science and Grid Computing*, pp. 8-pp, 2005.
3. S. Kindermann, "Climate Data Analysis and Grid Infrastructures: Experiences and Perspectives," *Grid-Enabling Legacy Applications and Supporting End Users Workshop (GELA)*, vol. 20. 2006.
4. P. Mendez-Lorenzo, J. T. Moscicki and A. Ribon, "Experiences in the gridification of the Geant4 toolkit in the WLCG/EGEE environment," *IEEE Nucl. Sci. Symp. Conf. Rec.*, vol. 2, 2006.
5. K. Beck *et al.*, "Manifesto for Agile Software Development," 2012. [Online]. Available: <http://www.agilemanifesto.org/>. [Accessed 14 Feb. 2019].
6. L. Zhu, L. Bass and G. Champlin-Scharff, "DevOps and Its Practices," *IEEE Softw.*, vol. 33, n. 3, pp. 32–34, 2016.
7. P. Kunszt, "European DataGrid project: status and plans," *Nucl. Instr. Meth. Phys. Res. A*, vol. 502, no. 2, pp. 376–381, Apr. 2003.
8. G. Avellino *et al.*, "The DataGrid workload management system: Challenges and results," *J. Grid Comp.*, vol. 2, no. 4, pp. 353–367, 2004.
9. F. Gagliardi, B. Jones, M. Reale and S. Burke, "European DataGrid Project: Experiences of Deploying a Large Scale Testbed for E-science Applications," in *Performance Evaluation of Complex Systems: Techniques and Tools*, Performance 2002. LNCS, vol. 2459, pp. 480–499, 2002.
10. I. Foster and C. Kesselman, "Globus: a Metacomputing Infrastructure Toolkit," *Int. J. High Perfor. Comput. Appl.*, vol. 11, no. 2, pp. 115–128, Jun. 1997.
11. L. Momtahan and A. Martin, "e-Science Experiences: Software Engineering Practice and the EU DataGrid," in *Proc. 9th Asia-Pacific Softw. Eng. Conf.*, pp. 269–275, Dec. 2002.
12. T. Dingsoyr, S. Nerur, V. Balijepally and N. B. Moe, "A decade of agile methodologies: Towards explaining agile software development," *J. Syst. Softw.*, vol. 85, no. 6, pp. 1213–1221, Jun. 2012.
13. M. Paulk, B. Curtis, M. Chrissis and V. C. Weber, "Capability maturity model for software," *Softw. Eng. Inst.*, Technical Report CMU/SEI-93-TR-024, ESC-TR-93-177, Feb. 1993. [Online]. Available: https://resources.sei.cmu.edu/asset_files/TechnicalReport/1993_005_001_16211.pdf. [Accessed 14 Feb. 2019].
14. Quality Assurance Group, "DataGrid - European DataGrid Developers' Guide," 2003. [Online] Available: <https://edms.cern.ch/ui/file/358824/1.1/EDG-DevGuide-v1-2.pdf> [Accessed 14 Feb. 2019].
15. DataGrid, "DataGrid Internal Document - Quality and Performance Indicators for DataGrid," 2003. [Online] Available: <https://edms.cern.ch/ui/file/386039/2/QIv0-3.pdf> [Accessed 14 Feb. 2019].
16. *Enabling Grids for E-science (EGEE)* project, European Community Research and Development Information Service (CORDIS). [Online] Available: http://cordis.europa.eu/project/rcn/80149_en.html [Accessed 14 Feb. 2019].
17. *Enabling Grids for E-science-II (EGEE-II)* project, European Community Research and Development Information Service (CORDIS). [Online] Available: http://cordis.europa.eu/project/rcn/99189_en.html [Accessed 14 Feb. 2019].
18. *Enabling Grids for E-science-III (EGEE-III)* project, European Community Research and Development Information Service (CORDIS). [Online] Available: http://cordis.europa.eu/project/rcn/87264_en.html [Accessed 14 Feb. 2019].
19. F. Gagliardi and M. E. Begin, "EGEE – providing a production quality grid for e-science," in *2005 IEEE Inter. Symp. Mass Storage Syst. Technol.*, pp. 88–92, Jun. 2005.
20. T. Ferrari *et al.*, "Resources and services of the EGEE production infrastructure," *J. Grid Comp.*, vol. 9, no. 2, pp. 119–133, 2011.
21. E. Laure *et al.*, "Programming the Grid with gLite," *Computational Meth. Sci. Technol.*, vol. 12(1), pp. 33–45, 2006.

22. D. Thain, T. Tannenbaum and M. Livny, "Condor and the Grid," in *Grid Computing: Making the Global Infrastructure a Reality*, ch. 11, pp. 63–70, 2003.
23. *Definition and Documentation of the Revised Software Life-Cycle Process*, Milestone MSA3.4.2, 2010, EGEE-III project. [Online]. Available: https://edms.cern.ch/ui/file/1062487/2/EGEE-III-MSA3.4.2-1062487-v1_4.pdf. [Accessed 14 Feb. 2019].
24. A D Meglio, M-E Begin, P Couvares, E Ronchieri and E Takacs, "ETICS: the international software engineering service for the grid," in *J. Phys.: Conf. Ser.*, vol. 119, no. 4, 042010, 2008.
25. C. Aftimiei *et al.*, "Towards next generations of software for distributed infrastructures: The European Middleware Initiative," in *2012 IEEE 8th Inter. Conf. on E-Science*, Chicago, IL, pp. 1–10, 2012.
26. *ISO/IEC 9126 Software Engineering - Product Quality*, International Organization for Standardization. [Online]. Available: <https://www.iso.org/standard/22749.html>. [Accessed 14 Feb. 2019].
27. M. Alandes *et al.*, "Experiences with Software Quality Metrics in the EMI middleware," in *J. Phys.: Conf. Ser.*, vol. 396, no. 5, 052003, 2012.
28. I. C. Plasencia, "EGI.eu the European grid initiative," in *Proc. 4th Iberian Grid Infra. Conf.*, pp. 5–15, 2010.
29. H. Cordier *et al.*, "From EGEE Operations Portal towards EGI Operations Portal," in *Data Driven e-Science (ISGC2010)*, pp. 129–140, 2011.
30. T. Antoni, *et al.*, "Global grid user support—building a worldwide distributed user support infrastructure," *J. Phys.: Conf. Ser.*, vol. 119, no. 5, 052002, 2008.
31. G. Mathieu and J. Casson, "GOCDB4, a New Architecture for the European Grid Infrastructure," in *Data Driven e-Science (ISGC2010)*, pp. 163–174, 2011.
32. M. David *et al.*, "Validation of Grid Middleware for the European Grid Infrastructure," *J. Grid Comp.*, vol. 12, no. 3, pp. 543–558, 2014.
33. *EGI Quality Criteria*. [Online]. Available: <https://egi-qc.github.io/>. [Accessed 14 Feb. 2019].
34. *Engaging the EGI Community towards an Open Science Commons (EGI-ENGAGE)* project, European Community Research and Development Information Service (CORDIS). [Online] Available: http://cordis.europa.eu/project/rcn/194937_en.html [Accessed 14 Feb. 2019].
35. P. Orviz *et al.*, "umd-verification: Automation of Software Validation for the EGI Federated e-Infrastructure," *J. Grid Comp.*, vol. 16, no. 4, pp. 683–696, 2018.
36. D. Salomoni *et al.*, "Indigo-datacloud: a platform to facilitate seamless access to e-infrastructures," *J. Grid Comp.*, vol. 16, no. 3, pp. 381–408, 2018.
37. G. Casale *et al.*, "Current and future challenges of software engineering for services and applications," *Procedia Computer Science*, vol. 97, no. 3, pp. 34–42, 2016.
38. J. Gomes *et al.*, "Initial Plan for WP3," *INDIGO-DataCloud Deliverable 3.1*. [Online]. Available: <https://www.indigo-datacloud.eu/documents/initial-plan-wp3-d31>. [Accessed 14 Feb. 2019].
39. Pablo Orviz *et al.*, "A set of common software quality assurance baseline criteria for research projects," 2017. [Online]. Available: <http://hdl.handle.net/10261/160086>. [Accessed 14 Feb. 2019].
40. Members of the INDIGO-DataCloud, DEEP Hybrid-DataCloud and eXtreme DataCloud collaborations, 2015-2020, "A set of Common Software Quality Assurance Baseline Criteria for Research Projects," 2018. [Online]. Available: <https://github.com/indigo-dc/sqa-baseline> [Accessed 14 Feb. 2019].
41. Paul Hamill, "Unit test frameworks: tools for high-quality software development," *O'Reilly Media, Inc.*, 2004.
42. indigo-dc, "indigo-dc Spaces - GitBook," 2018. [Online]. Available: <https://www.gitbook.com/@indigo-dc>. [Accessed 14 Feb. 2019].
43. puppetforge, "Modules by INDIGO Datacloud - Puppet Forge," 2018. [Online]. Available: <https://forge.puppet.com/indigodc>. [Accessed 14 Feb. 2019].
44. GALAXY, "Ansible Galaxy," 2018. [Online]. Available: <https://galaxy.ansible.com/indigo-dc/> [Accessed 14 Feb. 2019].
45. GitHub Developer, "GitHub API v3 — GitHub Developer Guide," 2018. [Online]. Available: <https://developer.github.com/v3/> [Accessed 14 Feb. 2019].

46. Jenkins, “Jenkins,” 2018. [Online]. Available: <https://jenkins.io/> [Accessed 14 Feb. 2019].
47. GRIMOIRELAB, “GrimoireLab - Software Development and Community Analytics Platform,” 2017. [Online]. Available: <http://grimoirelab.github.io/> [Accessed 14 Feb. 2019].
48. Jenkins Indigo-dc, “Jenkins - Indigo-DataCloud,” 2018. [Online]. Available: <https://jenkins.indigo-datacloud.eu:8080/> [Accessed 14 Feb. 2019].
49. Indigo-dc, “indigodatacloud - Docker Hub,” 2018. [Online]. Available: <https://hub.docker.com/u/indigodatacloud> [Accessed 14 Feb. 2019].
50. GitHub’s indigo-dc organization, “indigo-dc,” 2018. [Online]. Available: <https://github.com/indigo-dc> [Accessed 14 Feb. 2019].
51. D. M. Rafi *et al.*, “Benefits and limitations of automated software testing: Systematic literature review and practitioner survey,” in *Proc. 7th Int. Workshop Automation Softw. Test*, 36–42, 2012.
52. GitHub, “Learn Git and GitHub without any code!,” 2018. [Online]. Available: <https://github.com/> [Accessed 14 Feb. 2019].
53. L. Chen, “Continuous delivery: Huge benefits, but challenges too,” *IEEE Softw.*, vol. 32, no. 2, pp. 50–54, 2015.
54. G.C.P. Van Zundert and A.M.J.J. Bonvin, “DisVis: quantifying and visualizing the accessible interaction space of distance restrained biomolecular complexes,” *Bioinformatics*, vol. 31, no. 19, pp. 3222–3224, 2015.
55. G.C.P. Van Zundert and A.M.J.J. Bonvin, “Fast and sensitive rigid-body fitting into cryo-EM density maps with PowerFit,” *AIMS Biophys.*, vol. 2, no. 20150273, 73–87, 2015.