Framework and tooling proposals for Agile certification of safety-critical embedded software in avionic systems

Claude Baron, Vincent Louis

Framework and tooling proposals for Agile certification of safety-critical embedded software in avionic systems

***Abstract*—The abstract must be between 150–250 words.**
This article focuses on quality improvement in the development of DO-178C/ED-12C safety-critical software, optimizing development costs through the implementation of a "continuous certification" integral process. We discuss the major problems associated with traditional, V-cycle software development practices, and highlight the many advantages of adopting a "continuous certification" process based on a more Agile approach. The article proposes a framework for deploying this process, explaining the development of the framework and demonstrating its compliance with the requirements of certification standards. We also propose a tooling based on open-source, off-the-shelf solutions to implement the framework and illustrate its efficiency and effectiveness on an industrial case study.

***Index Terms*— Embedded software, software engineering, software safety, computer-aided software engineering, Agile software development, certification, continuous production**

Other keywords: DevOps, Processes, Methods and tools, Continuous certification, Agility, Test-driven development, Continuous integration, Test automation.

I. INTRODUCTION

Developing avionics software subject to certification requires a thorough examination in order to deem its implementation acceptable. The certification process is essential in a safety-critical system, in which a failure may result in injuries or the loss of human life. Independent authorization for implementing this type of system is made necessary by the judiciary nature of our society. The societal need to ensure and guarantee the safety[1] of individuals entails the implementation of a number of actions, which can prove difficult if the associated challenges are not taken into account from the outset of the project.

This article expands on [Baron 2021] by providing an efficient and structured methodological approach to natively address specific certification requirements. It aims to enhance, fast-track and streamline the development of avionics software subject to certification. Our framework promotes a better integration of certification requirements by accelerating software development without compromising compliance with safety requirements. The framework relies on the principles of continuous development that—combined with Agile project management—enables the iteration and automation of certification actions in a transparent manner and with limited impact on the overall development cost.

Section II sets out the background to this study and introduces the research problem. Section III describes our framework and tooling proposals for continuous certification. In Section IV, the proposal is exemplified on a case study. The article concludes by discussing the advantages and limitations of our proposal.

II. BACKGROUND AND RESEARCH PROBLEM

A. Certification process and its role in aeronautics

Safety is the primary concern in the aircraft industry. A high level of reliability of the aircraft and all its systems, parts and equipment must be demonstrated through a process called certification. The certification process ensures a high level of safety for the industry and a low accident rate for society. Since it defines regulatory requirements imposing constraints on aircraft development, companies must demonstrate that they have implemented monitoring and inspection measures [Hilderman 2009], which determine whether an aircraft can obtain a type certificate. The stakes are thus commercial—since marketing an aircraft without a type certificate is not possible—but above all societal because manufacturers commit to ensuring the safety of passengers on board, ground crew and infrastructure.

Major players in aviation safety include the European Union Aviation Safety Agency (EASA) for civil aircraft and

[1] The concept of safety focuses on preventing accidents and minimizing the consequences of unforeseen accidents. This includes injuries or the loss of human life, property damage, mission failure, and environmental damage [Laprie 1996]. Ensuring safety requires a planned, disciplined, and systematic strategy for monitoring, identifying, analyzing, evaluating, and eliminating hazards throughout the life cycle of a system so as to prevent or reduce the number of accidents [Leveson 2003].

the Directorate General of Armaments (DGA) for French military and government aircraft. EASA has been the European aviation authority since its creation in 2002, exercising responsibilities that include drafting aviation safety laws, providing technical advice to the European Commission and European Union (EU) member states, certifying aircraft airworthiness, and approving global aeronautical design organizations within and outside the EU. With respect to the field of military aeronautics, the DGA has been responsible for type certification[2] and airworthiness monitoring[3] of state aircraft since 2006 [Louis 2019].

When a system function is deemed safety-critical[4], these authorities demand that the framework used for the system development process has been proven to be acceptable. Operating aircraft anywhere in the world requires certification by an authority. The certification process entails demonstrating compliance with all regulatory requirements for each aircraft type to a certification authority.

The certification process applies to aircraft, engines and propellers. Certification authorities treat software as part of the embedded system or equipment installed on the certified product [EASA CS-25 2018]. In other words, the software is not certified as a stand-alone product. All systems and equipment—including embedded software—must be approved before being accepted for certification.

Only the proper application of an engineering process can ensure that the product fulfills the safety objectives. Audits are conducted to verify the technical content generated by the implemented processes. Whether the certification authorities approve the product depends on the successful demonstration or test of the product life cycle. For the development of systems with software-supported functions, specific objectives must be applied to enable the implementation of a "software certification" [Baron 2021].

B. Standards and requirements for software certification in aeronautics

Companies seeking to comply with existing regulations must take into consideration numerous normative good practice guides and recommendations, which result from a consensus in the aeronautical community and contain the best shared practices. As the main regulatory bodies (e.g. EASA and the FAA[5]) acknowledge their compliance with the regulations, these guides have become the main reference and are considered as de facto standards.

Thus, aviation standards such as ARP 4754A [ARP4754A 2011], ARP 4761 [ARP4761 1996], DO-178C [RTCA DO-178C 2012] and DO-254 [RTCA DO-254 2006] represent a classic approach to managing development in the aviation industry by taking into account certification requirements.

The DO-178C/ED12C standard is the reference for the development of embedded software in aeronautics. It defines five software criticality levels called DAL (Development Assurance Level), ranging from A (the most critical) to E (the least critical):

- DAL A: An error in the software would lead to a "catastrophic" failure (preventing the safe continuation of the flight or the landing);
- DAL B: An error in the software would lead to a "hazardous" failure (large reduction in functional capabilities or safety margins, physical distress or excessive workload increase for the flight crew, serious injuries to a small number of passengers or crew, etc.);
- DAL C: An error in the software would lead to a "major" failure (significant reduction in functional capabilities or safety margins, significant workload increase for the flight crew, physical distress possibly including injuries to passengers, etc.);
- DAL D: An error in the software would lead to a "minor" failure (slight reduction in functional capabilities or safety margins, slight workload increase for the flight crew, physical discomfort for the passengers, etc.);
- DAL E: An error in the software would lead to a "no safety effect" failure on the operational capabilities of the aircraft or on the workload of the pilot.

The closer the DAL level is to A, the higher the number of requirements will be. DAL E-level software (once the Certification Authority has confirmed its status) is deemed outside the scope of DO-178, i.e. no requirements are imposed.

[2] A type certificate is specific to a particular category of aircraft. The certificate attests that the aircraft is built according to an approved "type" manufacturing design and in compliance with airworthiness requirements.

[3] Airworthiness is defined as the ability of an aircraft (irrespective of type: airplane, helicopter, glider, drone, etc.) to operate under acceptable safety conditions with respect to passengers and crew, overflown populations and other airspace users.

[4] A software program is considered safety-critical if its failure would significantly affect the safety or existence of people, companies or property. The criticality of the functions provided by a system is determined by the failure conditions related to those functions.

[5] Federal Aviation Administration. Aviation authority that regulates civil aviation in the United States.

The DO-178C standard also specifies technical activities in the development process and verification activities designed to detect errors preemptively. It outlines the required steps to demonstrate that the software correctly implements the expected functions. For each process, the DO-178C defines: assurance objectives (e.g. defining the architecture and elements that enable coding) and the means to satisfy them, input data (e.g. specifications, development plan, design rules), activities (e.g. defining the architecture, derived requirements), products (e.g. the design description), and transition requirements. It is worth noting that DO-178C does not impose a method or tool. Instead, it sets objectives to be attained, allowing the various actors to find the most effective means of meeting them.

Since the standards are numerous and each one uses a different approach to describe its requirements, understanding and mastering the standards is complicated. Some companies adopt the DO-178C standard as their main reference, even if the standard is particularly strict and requires the implementation of rigorous processes to demonstrate compliance with the various objectives. Companies often struggle to comply with this standard, often due to financial or staffing constraints [Hilderman 2009].

To assist companies in meeting the certification requirements, the Digital Embedded Systems Engineering and Dependability Unit of the DGA TA—acting as a certification authority—has developed a reference framework [NT DGATA 2016] that explicitly covers the essential requirements for software engineering. These correspond to the non-negotiable, minimum requirements of an audit that must be observed. This reference framework defines the core requirements—organized into six areas (see Table 1)—to assist companies in strictly complying with them, primarily drawing on the applicable standards for the certification of embedded software, the DO-178C standard [RTCA DO-178C 2012] for aeronautics, the IEC 61508 standard [IEC 61508 2010] for industrial electronics, and the European Cooperation for Space Standardization (ECSS) space standards (in particular the one that addresses software product assurance [RNC-ECSS-Q-ST-80 2017]). We have adopted this reference framework in the article.

Table 1 – Areas covered in the DGA TA reference framework

Organization requirements
- Planning of development activities and cross-functional processes
- Compliance with security requirements derived from the security analysis
Requirements management
- Writing of High-Level Requirements (HLR) – software specification
- Description of software architecture
- Writing of Low-Level Requirements (LLR) – software design – detailed requirements
Conducting reviews
- Requirement reviews (HLR, architecture, LLR)
- Source code reviews (compliance with requirements, adherence to coding rules)
- Test procedure reviews
- Test results reviews
Verification activities
- High-level requirement testing (functional coverage)
- Integration tests
- Unit tests
Traceability
- System specifications – Software specifications
- Software specifications – Detailed design
- Detailed design – Source code
- Software specifications – Software testing
- Detailed design – Unit tests
Independence For the most safety-critical software (DAL A and DAL B), the DO-178C standard adds the notion of independence to ensure an objective evaluation of the completeness of an activity. Development and verification activities must therefore be entrusted to independent teams. Two types of independence are considered: • Independence between the reviews or analyses: the person who conducts the review must be different from the person who produced the data; • Independence between the individuals performing the activities: for example, between the coder and the person who selects the requirement-based test cases.

Figure 1 illustrates the standard steps of the software engineering process, encompassing the different levels of engineering (from the software and the software components up to the code). The different activities (e.g. the reviews) are found at each engineering level (from the software specification review and the detailed software requirements review to the code review). This figure adds the items from Table 1, placing them in relation to the different steps of the software engineering process.

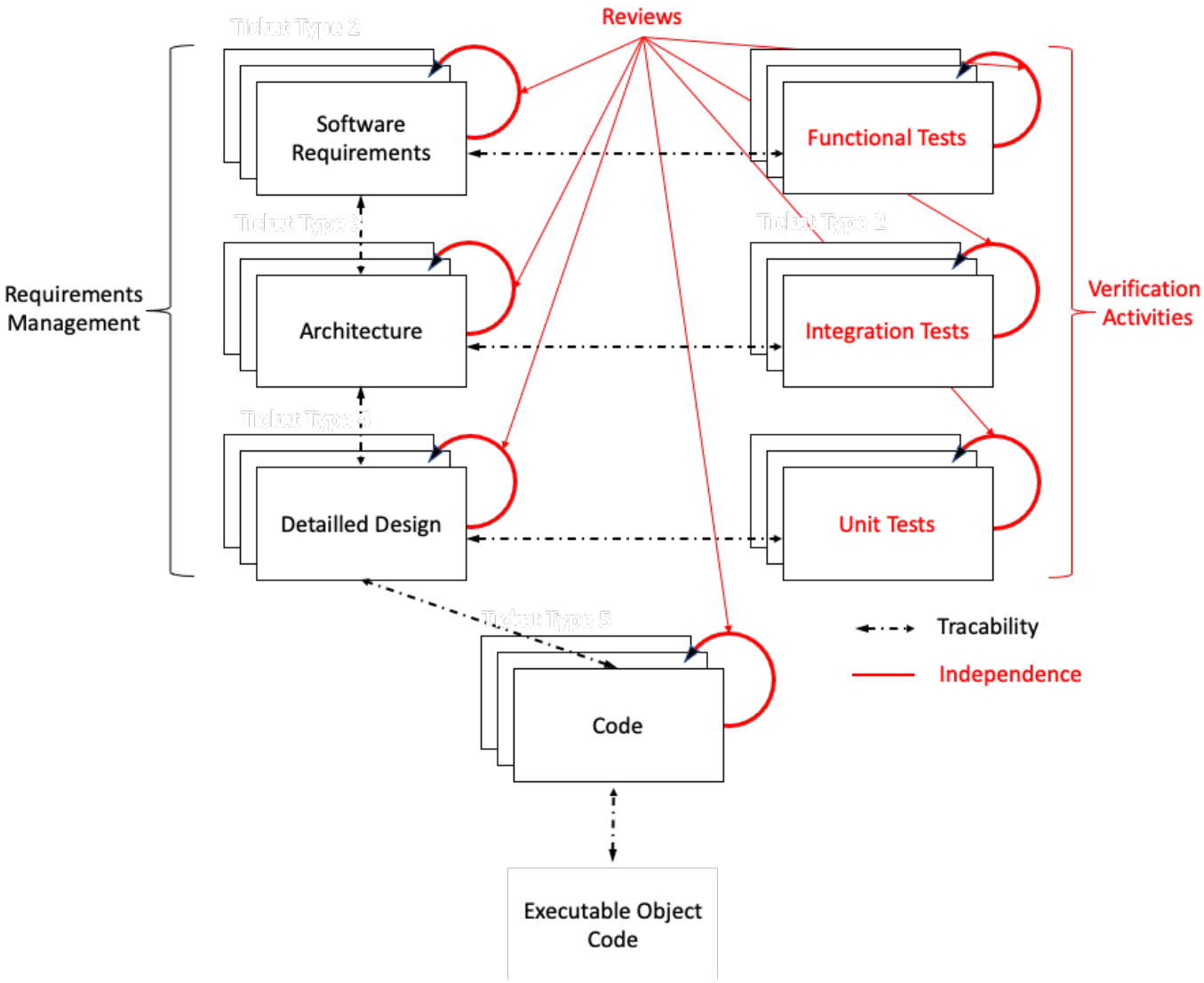

Figure 1 – Standard activities and certification requirements in software engineering

For each engineering activity shown in a box in

Figure 1, Table 2 provides an excerpt of the requirements described in the DGA TA reference framework. Each requirement receives a specific tag in the form [Ni-SWj], where *i* (from 1 to 3) indicates the criticality level and *j* the ID number of the software requirement at each level. It is worth noting that the engineering level of the detailed software design is required only for level 2 critical functions (in the DGA TA reference framework).

Table 2 – Excerpt of requirements from the DGA TA reference framework for different engineering activities

Engineering activities	**Excerpt of requirements from the DGA TA reference framework**
Specification	[N1-SW1] The contractor shall develop a software specification in the form of requirements and with the following characteristics: Uniquely identifiable, non-ambiguous, complete, verifiable, consistent and traceable.
Functional tests	[N1-SW9] For software verification, the contractor shall develop a set of test cases to functionally cover 100% of the requirements from the software specification, as well as all interfaces between software components.
Architecture definition	[N1-SW3] The contractor shall develop and apply a general software design detailing the static and dynamic architecture including the interfaces between software components, the hardware architecture hosting the software, and the allocation of software specification requirements to software components.
Detailed design	[N2-SW3] For each software component, the contractor shall develop a software specification in the form of requirements and with the following characteristics: Uniquely identifiable, non-ambiguous, complete, verifiable, consistent and traceable. Derived requirements shall be identified as such and accounted for.
Unit tests	[N2-SW6] The contractor shall automate the code coverage analysis and attain a 100% coverage rate. Automating test execution on the software specification is recommended, complementing if necessary with unit tests to attain the required structural coverage rate on all instructions.
Coding	[N1-SW4] Define coding rules for each programming language used, according to a well-known standard Specify the safety-criticality level of each rule (blocking, critical, major, etc.) Automate rule verification [N1-SW5] Apply and continuously monitor these coding rules
Integration tests	[N1-SW10] Evaluate the performance of time-related aspects. Evaluate the management of material resources. Verify the external interfaces using the validity/invalidity of values and robustness (load, initialization, etc.)

C. Current practices and limitations

Certification rules in aeronautics ensure a high safety level on the air transport market worldwide. While they provide societal guarantees, they also complicate the development of systems, affecting the technical, economic and organizational components of a company.

Today, the major manufacturers in the aeronautics sector conduct highly organized aircraft certification activities [Rempel 2014]. Nevertheless, regulations are still perceived as difficult to implement by players in the aeronautics sector—especially newcomers and SMEs—since they must meet the requirements of all market stakeholders from equipment manufacturers to airlines and other agents in the supply chain [Lemoussu 2018]. Misinterpretations or late consideration of the certification requirements are frequently observed. Thus, certification regulations are often dealt with at the end of development where additional activities are performed, only to prove that the process is standard-compliant [Louis 2019]. Indeed, software development stakeholders usually try to postpone certification activities at the end of the project thinking that they are reducing costs, which is a risky approach since software issues could be discovered very late. They try to escape V-Cycle inflexibility by delaying certification activities to avoid a very late system validation after a first delivery. Demonstrating compliance therefore requires additional effort, which occasionally proves to be very costly — if only because of unavoidable redesign activities.

The current economic challenge consists in avoiding situations leading to late completion of certification-related activities, on the grounds that they do not bring any added value: once the software has been delivered to the customer and the acceptance tests have been successfully completed, the proof of a requirements review is no longer of interest. Conducting this review during development (when the data is produced) is much more interesting, to limit the effort on the one hand and to maximize the impact of anomaly detection (completeness, testability, etc.) on the other. Striving for compliance with regulatory requirements on an ongoing basis helps guard against this risk [Steghofer 2019], as the interest of conducting certification activities (if they are conducted too late) will always wane alongside economic pressure.

A more precise analysis would highlight certain limitations, leading to consider avenues for improving current industrial practices in the development of embedded software in avionics. These limitations mainly stem from a highly linear development process that is conducted with insufficient interaction with the customer and with little room left for changes in the initial requirements. In the past, most companies have changed their software development process to a Model Based Development (MDB) process. While this approach is still justified for controlling the design of complex algorithms such as engine control law or flight control system, it is not a cost-effective solution for major software applications for two reasons. First, DO178C supplement related to MBD transposes the activities to the model level and does not simplify the process, a lot of activities are needed to confirm that the model is correct and complete. Second, modern software workflows natively integrate most of the techniques for performing the activities required for a safety-critical software (traceability, unit testing, code coverage, independence...). It is also easy to recruit already trained master students.

The traditional V-cycle model remains a reference for the development of embedded software in aeronautics. It describes a set of sequential processes aimed at successively generating the application specifications, the specification document, the general design document, the detailed design document, test plans at various stages, and the application itself [Ninni 2021]. In this development model, the project requirements and deliverables are defined at the beginning of the project and each phase must be completed to proceed to the next phase [Balaji 2012]. Many variants of the V-cycle exist, which are made more flexible by introducing iterative cycles at the level of each process or between processes. In any case, the framework of a development project is roughly the same — what we want to build is defined with precision from the outset before undertaking the actual implementation.

That being said, in the case of aircraft programs—where development time is essential—requirements frequently change during the course of the project and it is not surprising that specifications are refined or even reassessed several times along the way. The further into the project the customer is introduced to the first versions of the software, the more difficult these requirement changes will be to implement. If no regular exchanges are held with the customer during development, the product may gradually diverge from their expectations (which usually vary from the needs initially expressed). The emergence of new technologies is also common during development, creating many opportunities to consider in the software design. Should these innovations be overlooked, the software could be at odds with the industrial trends of the market at the time of its delivery. Last but not least, the standards themselves—including certification standards—also evolve throughout development.

Because of these difficulties, the skid diagnosis is often the same: construction stalls after rigorous specifications and a thorough design and the initial project choices get reassessed. This entails revising entire sections of the design and a considerable loss of time, a major cost overrun and sometimes a significant quality loss in the final product, insofar as the proposed design incorporates the necessary adaptations relatively well.

Various efforts have been attempted to overcome these problems; thus, modeling methods and tools constantly evolve in search of greater efficiency (expression precision, formal expression of functional requirements using tools such as Stimulus, etc.). Nevertheless, these proposals do not have a significant impact on the overall progress of projects beyond ad hoc improvements on certain activities. It is worth pondering whether the solution to these problems would not arise from conducting a broader effort on the development process, instead of placing even greater emphasis on the completeness of the initial phase.

D. Perspectives for improvement and objectives of the study

Some companies use the Agile development approach, which is widespread in software development, has well-documented benefits, and is of increasing interest even beyond the scope of software development [Safe 2021]. Agile approaches rely on organizing projects into short development cycles, thus avoiding the characteristic tunnel effect of linear development. They seek to deliver intermediate versions of operational solutions to the client so that they can measure the progress of the project and validate its course, thus reducing the risks and costs of development.

Yet, very few players in the field of aeronautics use Agile methods to develop embedded software subject to certification. While the rationale is often understandable, the choice of not embracing Agile deprives them of the benefits of these approaches. Some are reluctant to move away from a traditional V-cycle approach, which they deem to be recommended by the DO-178C standard. Others prefer to use a proven method already validated by the certification authority, since they are concerned about having to convince said authority to implement a new method [Baron 2021]. Some others do not upgrade their engineering workshop for fear of having to demonstrate that existing projects are not affected by the upgrade. The concern, which is legitimate, is that the transition to an Agile development will prevent complying with the requirements linked to the certification objectives.

Progressively incorporating compliance with these requirements throughout an Agile development process would save precious time, since certification is a mandatory process that can considerably lengthen development time if not properly addressed from the outset of the project [DoD 2018]. Some aeronautics and automotive companies have already led successful attempts to develop Agile-based critical embedded software, showing interest in improving practices if not starting a trend.

Thales Avionics demonstrated the usability of Agile for the development of the ADIRU calculator [Chenu 2013]: the customer observed 99% fewer errors when compared to a similar project, the number of residual defects found by the end user went from 10 per KLoC to 0.07 per KLoC, the cost of product integration was decreased from 30% to nearly 5% of the project budget, and the cycle time was reduced from one year to 20 days. In a 3-year timeframe, nine versions of the product were delivered on schedule to the customer, and the software successfully passed its first flight-tests. [Chenu 2022] shows how Thales Avionics keeps improving their configuration management and continuous integration practices and tools in order to gain in rigor, traceability and efficiency, the concept is to manage all engineering data in source code form.

Airbus Helicopter experimented with the Scrum framework [Scrum 2018] to develop military embedded software, showing there are no actual contradictions between Agile practices and avionics software certification objectives. Surveying several Airbus projects, [Marsden 2018] and [Kuehne 2020] outlined that significant improvements in quality, schedule and cost were achieved by using Agile. They concluded that Agile is not only compatible with DO-178C but actually simplifies the latter through greater visibility and openness.

Tesla implemented a continuous process to develop safety-critical embedded software [Vöst 2016] that features Agile principles of continuous integration and continuous deployment.

Sogilis succeeded in developing a DO-178C/ED12C – DAL A level “Autopilot” software for drones by applying certain Agile principles, test-driven development in particular [Mrabti 2018].

US DoD report [LaPlante 2018] regarding more than 29 studies found that agile methods and DevOps workflow yielded the following return on investment:

- 29% lower cost,
- 91% better schedule,
- 50% better quality,
- 400% better job satisfaction.

These experiments illustrate a current industrial trend in safety-critical development that consists in rendering the certification process even more integral to the development process. They also demonstrate that safety assessments can be continuously performed by the certification authorities.

Analyzing the limitations of current practices and of these successful experiments, [Baron 2021] demonstrated the feasibility of combining Agile practices and certification. Agile frameworks foster communication within and between teams, as well as with stakeholders, customers and certification authorities (who can be considered as customers). Their approach is remarkably pragmatic — the key concern is that everything works properly and that everyone involved is satisfied, including the certification authorities. The challenge in the development of safety-critical embedded software is to reach a “certification ready” status by the end of each iteration for every piece of data or activity produced (Requirement, Test, Review).

Louis [2019] calls this approach “continuous certification”, which encourages the integration of certification requirements into continuous software development. This approach boosts the level of confidence by ensuring that the activities are conducted at the right time, rather than right before an audit just to comply with regulations. It simplifies certification audits by providing immediate traceability of all data and systematic evidence of the completion of activities required for certification (reviews, test results, structural coverage rate, performance metrics, etc.). Continuous certification is the culmination of a natural evolution of the continuous development process, integrating

certification activities before each deployment[6] [Fowler 2010] [Gaudin 2013] [Humble 2010], thus combining methodology and team values while leaving plenty of room for the choice of development tools.

This article expands on [Baron 2021] by proposing an Agile framework and a set of tools adapted to the certification requirements.

III. FRAMEWORK PROPOSAL

Our proposal consists in defining a "hybrid" framework that enables the application of different Agile principles to the development of critical software. To that end, we used as inspiration some concepts from the most recognized Agile frameworks, to which we have associated a ticket-based project management.

A. Agile principles chosen to build the proposal

In 2001, the Agile Manifesto [Beck 2001] determined four essential values in software development and 12 principles underlying these values, and adopted the term "agile" to refer to them [Thummadi et al., 2011]. Agile practices employ an iterative, incremental and adaptive approach that focuses on the autonomy of the human resources involved in the specification, production, and validation of an integrated and continuously tested application [Rahman, 2015]. Development is conducted in short cycles with the purpose of continuously delivering operational software; there is greater involvement of all stakeholders in the development process and they follow the progress of the project regularly [Edeki, 2015] [Kumar and Bhatia, 2012]. Thus, Agile corresponds to an approach to collaboration in software development and workflows.

Agile rapidly gained momentum with the increasing popularity of frameworks such as Scrum, Kanban, DevOps or eXtreme Programming (XP), to mention only the most popular [Stellman and Greene, 2013].

Each of these frameworks offers interesting features in terms of software certification but also carries some limitations.

Our proposal integrates several concepts extracted from the most recognized Agile frameworks. We have chosen only those essential to the proposed solution. Beyond this selection, users may freely integrate any other useful concept into this framework proposal in accordance with their own strategy.

Drawing inspiration from Scrum, we propose a framework with short sprints. The objectives are defined at the beginning of the sprint. Since they are achievable, it is possible to have a partially functional product at the end of each sprint so that the software is ready to be audited on the functional scope previously set. Each team member has a specific role. Evolution of customer needs is not only allowed but encouraged, with the purpose of meeting these needs as closely as possible during the development of the product.

As is the case in DevOps, the interaction between the engineers writing the source code and the operations engineers responsible for application deployment is crucial. Automating the workflow results in a more reliable and streamlined application deployment.

We also take the following ideas from eXtreme Programming:

- Pair programming: by regularly switching roles, the margin of error is decreased; working in pairs also facilitates technical exchanges that let collaborators share their experience. Moreover, this ensures the continuous review of completed code.
- Coding standards: a good understanding between all members of a project is key to avoid ambiguity.
- Simplicity is key: designing an application with simple mechanisms facilitates the maintainability, testability and auditability of the software.

As all three frameworks (Scrum, DevOps and eXtreme Programming) rely on continuous development, the concept of continuous development and all the activities that stem from it are also incorporated in our proposal.

[6] The term "continuous development" refers to the entirety of operations conducted, from the source code to the deployment of the application for end users. All these operations are performed in accordance with Agile principles and include continuous integration, continuous inspection, continuous delivery and continuous deployment.

Table 3 and

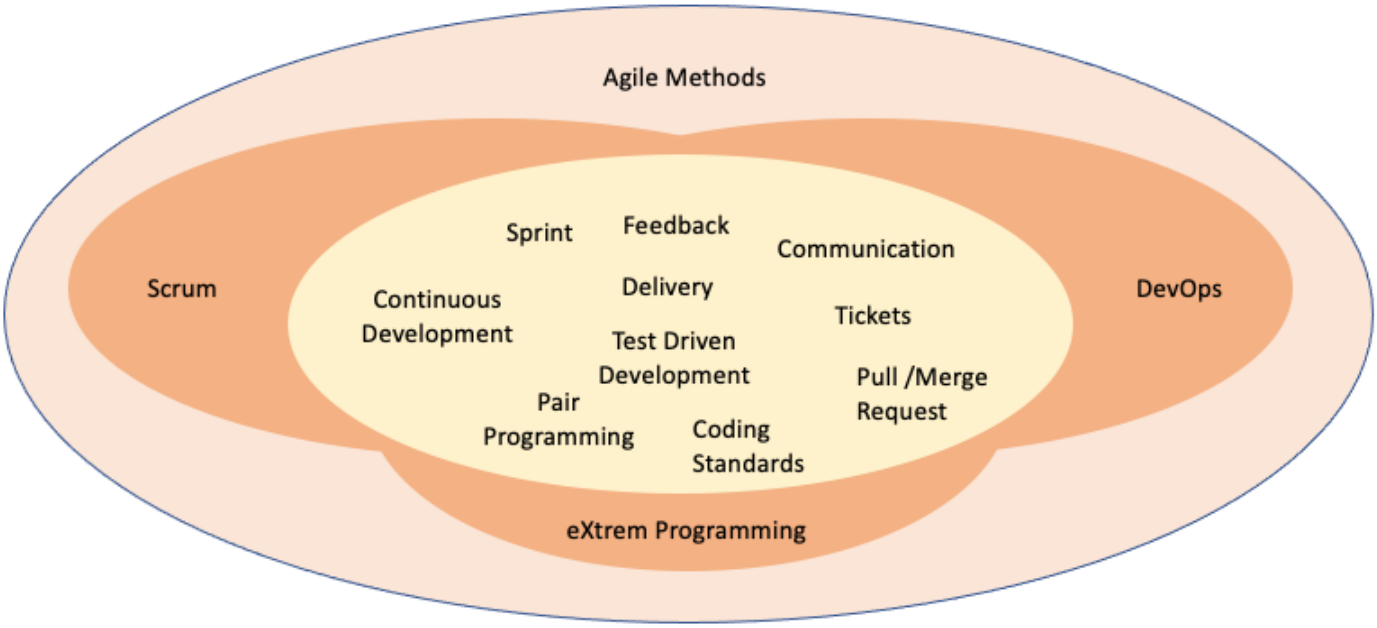

Figure 2 summarize the concepts and practices included in our framework proposal.

Table 3 – Concepts adopted from different Agile frameworks to build the proposal

Framework	**Concepts/practices**	**Compliance with essential certification requirements**
Scrum	Short sprints Achievable objectives Auditable software by the end of each sprint Each developer has a specific role and keeps it Functional needs may change	The certification requirements are met at each iteration. The needs of authorities are addressed at the beginning of each phase.
DevOps	Close interaction between the development team and the team dealing with the operational deployment of the software. Automation of all repetitive tasks until the application is deployed on the operational target.	A regular and incremental operational deployment prevents the big bang effect frequently seen with late integration.
eXtreme Programming	Pair programming Coding standardization Simplicity is key	Pair programming implies the systematic rereading of the data produced. Measuring algorithm complexity facilitates the maintainability, testability and auditability of the functions concerned.
Test-Driven Development	Development is driven by the tests	Test-driven development ensures that the tests are built according to the requirements (requirement-based testing).

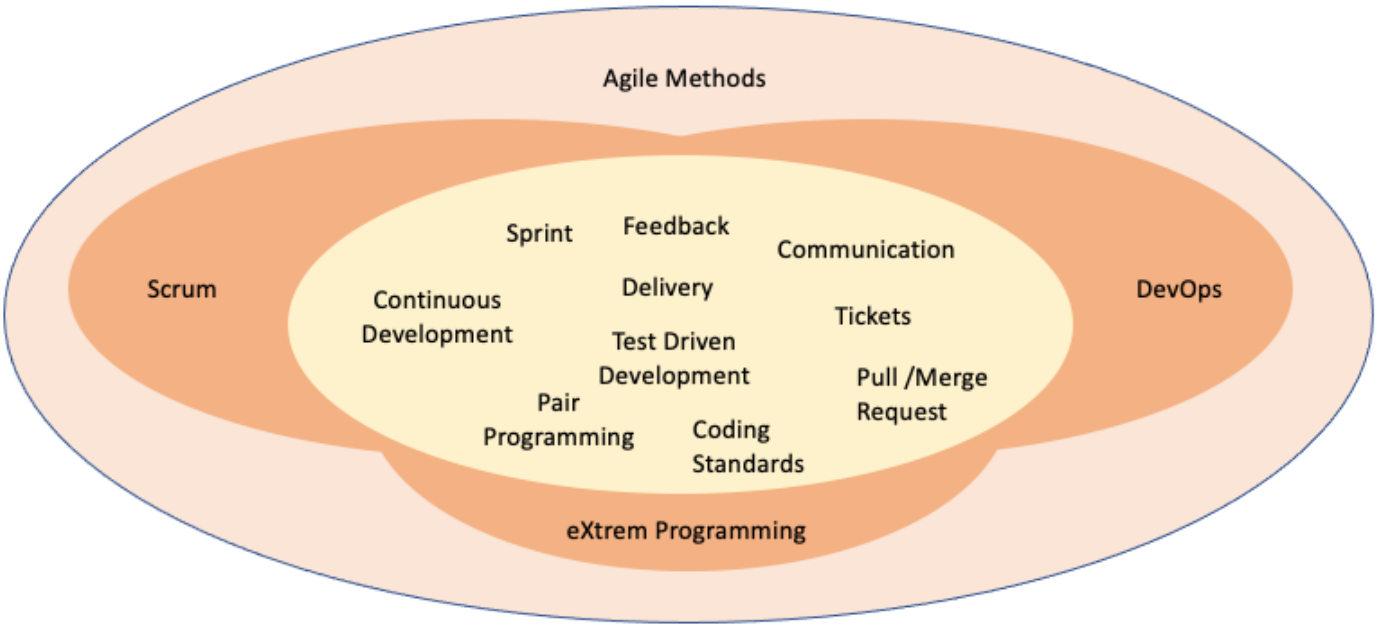

Figure 2 – Agile concepts included in the framework

This set of concepts ensures not only that the software "time to market" is reduced but also that the certification requirements are met at all times, thus guaranteeing the safe and reliable deployment of each new version. The implementation of these concepts facilitates the "auditability" of the development process to demonstrate compliance with certification requirements, either by an independent authority or by self-assessment [Gallina, 2018]. Correct implementation of each software feature can be demonstrated with the evidence of conducted reviews, traceability between all data, and automation of verification and deployment activities.

Nevertheless, the sole implementation of these Agile concepts does not deal with the challenge of certification, as standards set objectives but do not specify the means to achieve them. For example, a traceability link can be established through a naming convention, which is not necessarily an efficient way of achieving the objective. The implementation of other tools such as DOORS can be challenging when version managing certain data (particularly architecture diagrams) and when linking all levels of engineering (requirements, tests, code files).

Seeking to improve the efficiency of the practice and to propose adapted means, we built the proposal as a ticket-based project management. This type of management—nowadays standard in IT projects—allows to list, track and sort all the bugs, incidents, problems, requests, etc. in order to prioritize their solving. Managing the workload and assigning specific tickets (e.g. to the people who will be in charge of handling them) is also possible with this approach. Ticket-based management also allows for tracking task progress — once a user has submitted a ticket, the concerned individuals can take turns updating the progress of its resolution, other associated problems, and so on until completion. Lastly, most of the tools that support this type of management (such as Jira, Octopus, GitHub, GitLab or Redmine) offer the possibility of monitoring certain project indicators, such as the number of tickets in progress, the number of closed tickets, the estimated workload of each co-worker, etc.

The processing of tickets associated with the requirements of reference frameworks will therefore result in most of the certification constraints being addressed. For example, in response to the issue of traceability raised above, ticket-based management facilitates the implementation of digital links between activities, as well as tracking the contribution of each participant (editor, proofreader, etc.). The other underlying advantage of using tickets is the ability to trace the history of each ticket. This facilitates the configuration management of engineering data.

B. Implementation of a ticket-based project management

Some avionics software companies have their own versions of a ticket-based process, implementing different on-shelf tools or have defined some variant of the issue tracking process. The proposal we make in this article to adopt

this practice is just one possible option. We have chosen GitHub on the basis of its effectiveness, efficiency and widespread use in companies, but GitHub is only one possible option among those available on the market, that we chose to demonstrate the feasibility of the method.

The implementation of this type of management led us to define several types of tickets (numbered from 1 to 9 in Table 4). This classification can be tailored to the needs of the regulatory framework to which the project must conform.

Ticket types 2 to 6 are defined by extrapolating from the original purpose of a ticket-based management focused on the management of technical facts (type 8). These ticket types relate to the activities of the engineering process as described in

Figure 1, and are complemented by the definition of types 1 and 7–9. Type 1 refers to the continuous improvement of software engineering in an Agile approach, while types 7 and 9 concern certification activities. Specific processing of the software configuration parameter files is possible with type 7; type 9 covers all closeout activities for delivering the certified application. Type 9 tickets incorporate the declaration of compliance to the Software Accomplishment Summary in response to the certification plan (PSAC – Plans for Software Aspects of Certification, [RTCA DO-178C 2012]), data archiving and the required elements for restoring the executable, the identification and analysis of residual technical facts, and the identification of deviations from the PSAC.

Table 4 – Overview of the nine ticket types

Ticket type	Associated activity	Comments
1	Improvement of a process or a system	Evolution of certification plans, development standards or system requirements Artifact: update of the certification plan
2	Specification and Functional tests	Creation (writing based on system requirements) or review criteria (test cases) of software requirements
3	Architecture definition	Definition of software architecture
4	Detailed design and Unit tests	Definition of the expected behavior of the various components and test cases (unit tests) to validate their proper functioning
5	Coding	Writing of the source code of the components and the corresponding unit tests Execution of the test cases on the developed components
6	Generation of the executable	Compilation of the source code and execution of several test cases: integration tests, functional tests and user tests
7	Software configuration	Management of the different possible configurations of the software; ticket 7 is used to develop and verify new configurations Could be useful to distinguish between build time and runtime configurations
8	Open problems	Specific management of open problem reports
9	Delivery	All tests are rerun for final validation before a potential deployment

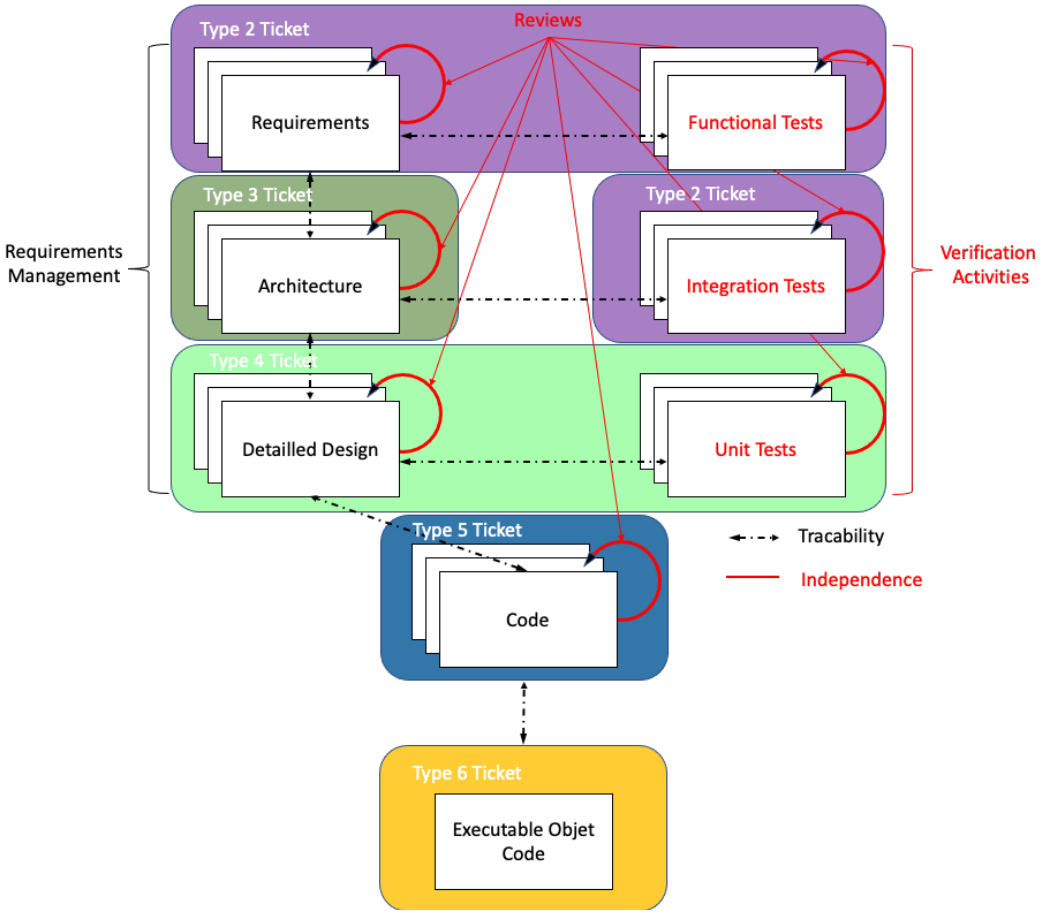

Figure 3 – Associating ticket types 2 to 6 to software engineering activities

Figure 3 reproduces the activities presented in Figure 1, indicating for each of them which type of ticket would be implemented. For example, the mauve type 2 tickets would describe all the functional requirements and their associated tests.

C. Ticket types and associated activities ensuring the requirements of the DGA TA reference framework

Defining these ticket types allows to deal with all the requirements of the DGA TA reference framework.

Table 5 lists the ticket types (color-coded to facilitate visualization within the figures) and their description (excerpted from Table 4), illustrating the requirements that can be associated with each one (excerpted from Table 2).

Table 5 — Description of ticket types and their relation to the DGA TA reference framework for requirements

Ticket type	Description	DGA TA reference framework for requirements
Type 1 : Process or System Improvement	***Process or system improvement*** *Evolution of certification plans, development standards or system requirements.*	[N1-SW17] The contractor shall provide a Software Development Plan and any applicable updates.

Type 2 : Software Requirement - HLR - HLT - IT	***Software requirements*** *Writing of software requirements and associated review criteria.* *Writing of software test cases.* *Definition of the structure and attributes of the data used for software configuration.*	[N1-SW1] The contractor shall develop a software specification in the form of requirements and with the following characteristics: Uniquely identifiable, non-ambiguous, complete, verifiable, consistent and traceable. [N1-SW9] For software verification, the contractor shall develop a set of test cases to functionally cover 100% of the requirements from the software specification, as well as all interfaces between software components.
Type 3 : Architecture	***Software architecture*** *The software architecture is defined using a type 3 ticket, on the basis of a first set of functions defined in the software requirements. As the requirements evolve, new ones are created.* *Definition of the dynamic architecture and the components.*	[N1-SW3] The contractor shall develop and apply a general software design detailing the static and dynamic architecture including the interfaces between software components, the hardware architecture hosting the software, and the allocation of software specification requirements to software components.
Type 4 : Component Requirement - LLR - UT	***Detailed design of the software*** *Definition of the expected behavior of the various components and test cases (writing of unit tests) to validate their proper functioning.*	[N2-SW3] For each software component, the contractor shall develop a software specification in the form of requirements and with the following characteristics: Uniquely identifiable, non-ambiguous, complete, verifiable, consistent and traceable. Derived requirements shall be identified as such and accounted for. [N2-SW6] The contractor shall automate the code coverage analysis and attain a 100% coverage rate. Automating test execution on the software specification is recommended, complementing if necessary with unit tests to attain the required structural coverage rate on all instructions.
Type 5 : Source Code	***Coding*** *Writing of the source code of the components and the corresponding unit tests.* *Execution of the test cases on the developed components.*	[N1-SW4] Define coding rules for each programming language used, according to a well-known standard. Specify the safety-criticality level of each rule (blocking, critical, major, etc.). Automate rule verification. [N1-SW5] Apply and continuously monitor these coding rules. [N1-SW6] Implement a technical debt measurement following the SQALE method by explicitly defining the evaluation criteria and the associated effort and thresholds. [N1-SW7] • Do not introduce a technical debt ratio > 5%. • No blocking or critical rule shall be transgressed without justification. [N1-SW8] Automate code coverage analysis and reach a rate of 80%. [N2-SW11] The contractor shall ensure that the static analysis metrics and their boundaries are respected: · cyclomatic number v(g) per function <= 10 · nesting depth of control structures <= 4 · inheritance tree depth <= 6 · copy/paste ratio <= 5% · number of effective lines per function (method, routine, etc.) <= 60 · number of source lines of code (SLOC) per software unit <= 1000 · comment/assertion ratio >= 20%

Type 6 : Integration - EOC generation	***Generation of the executable – Integration*** *Compilation of the source code, generation of the executable object code and execution of several test cases: integration tests, functional tests and user tests.*	[N1-SW9] The contractor shall: - develop a set of test cases to functionally cover 100% of the requirements from the software specification, as well as all interfaces between software components - perform the test procedures and record the test results (including structural coverage) [N1-SW10] • Evaluate the performance of time-related aspects • Evaluate the management of material resources • Verify the external interfaces using the validity/invalidity of values and robustness (load, initialization, etc.)
Type 7 : Configuration - PDI	***Software configuration*** *Management of the different possible configurations of the software; ticket 7 is used to develop and verify new configurations.*	[N1-SW15] Prior to conducting an activity, the contractor shall manage in configuration all the software engineering data used as input to the activity (development, tests, reviews, code generation, etc.).
Type 8 : Problem Report	***Open problems*** *Specific management of open problem reports.*	[N1-SW13] Implement a process for managing software open problems (documentation, processes, bugs, etc.) detected during development, industrial or public qualification, production, acceptance testing or in service (warranty period). [N1-SW14] Description and context of the open problem.
Type 9 : Delivery	***Delivery*** *All tests are rerun for final validation before a potential deployment.* *Note: Manual tests may be considered only at this stage if they are too costly.*	[N1-SW18] For each software version delivered, the contractor shall provide a Summary of Software Activities.

This table illustrates the requirements that can be associated with each ticket type. The set of ticket types does not cover all areas, although most of them are addressed. For example, creating a specific ticket for an audit is not suitable.

The life cycle of a ticket—which we shall describe in detail in section E—defines how these tickets are handled and the rules for their creation, ensuring that specific requirement characteristics are respected, such as the notion of independence required for conducting certain activities.

D. Ticket structure

The ticket system supports the engineering data specific to each development activity, such as the writing of requirements, test cases or source code.

In addition to specifying an activity, a ticket is also linked to one or more people responsible for the activity and its review, thus ensuring traceability of the actions and demonstrating the independence of the actors involved in each activity. Each ticket is tagged with a status that indicates the progress of the activity linked to the ticket.

The typical ticket structure is shown in Table 6, together with an example of a ticket.

Table 6 – Typical ticket structure and example of a ticket

Ticket structure	**Comments**	**Example of a ticket**
Ticket number	From 1 to xx	#13
Ticket type	From 1 to 9	Type 4 : Component Requirement - LLR - UT
Activity	Description of the activity linked to the ticket type (see Table 4)	*Definition of the expected behavior of the various components and test cases (writing of unit tests) to validate their proper functioning*

Ticket content	Technical description of the ticket	Signal should not be initialized in their declaration...
Author of the ticket	ID of the person in charge of the ticket	Vincent-Louis-DGA
Person in charge of the review	ID of the person reviewing the work conducted on the activity	Reviewer-DGA
Ticket status	*opened, to do, assigned, in progress, in review, verified, reopened, rejected, done* (see Figure 5)	Done

Figure 4 shows the structure of a type 4 ticket. It includes the ticket title, ticket number, ticket type, author of the ticket, person in charge of the review and the status of the ticket, as well as standard information such as the date of the actions conducted on the ticket.

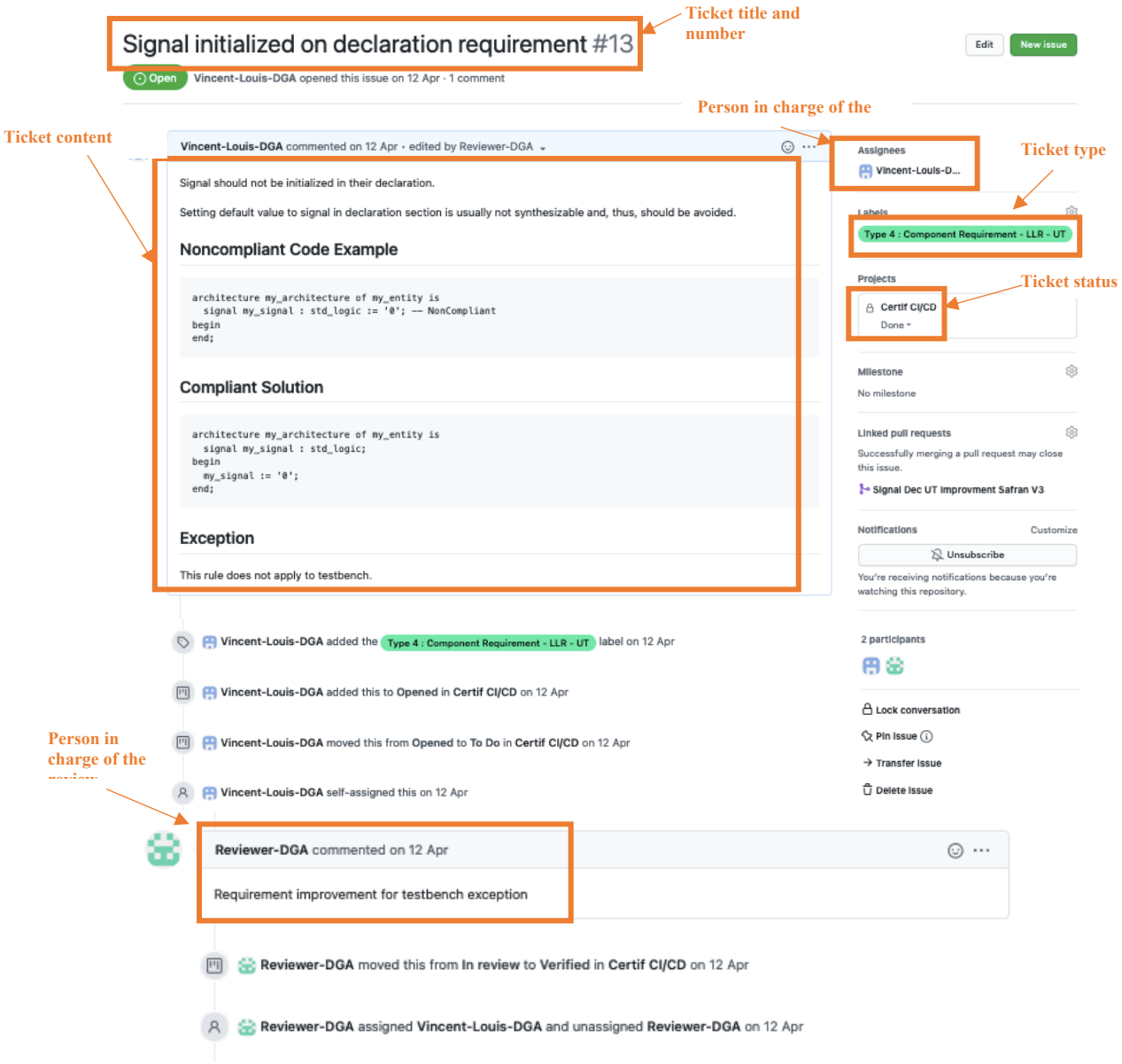

Figure 4 – Structure of a type 4 ticket

E. Ticket life cycle

A ticket can be created at the beginning or at any point in the project. When a ticket is created, the fields of the ticket structure are filled in (see Table 6) and a number is automatically assigned by the tool.

Traditionally, ticket-based management is used by bug tracking tools to collect, track and process open problems (such as Jira, Redmine, GitHub, etc.). The approach we propose here considers that ticket-based management can be

used for creating and managing all engineering data as soon as they are generated, be it at the beginning or in the course of the project.

Editing a ticket can result in the creation of other tickets linked to the first one. These links facilitate traceability between the different tickets (it is possible, for example, to link tickets managing engineering data at different levels, so that a functional HLR can be divided into several detailed design LLRs).

As shown in Table 6, a ticket can be found in one of several different statuses. The life cycle of a ticket is defined by the set of successive statuses a ticket takes; this is formalized in a workflow that describes the evolution model of the ticket status. This workflow is defined as a sequence of statuses linked to one another by transitions, which describe the required conditions for a ticket to move from one state to another. Therefore, a transition is directly linked to the actions performed in response to the activities described in the ticket.

The core premise of the workflow is to ensure that all tickets—regardless of their type—follow a sequence of statuses, so that the related activities are conducted in accordance with the requirements of the development process and team organization. It is very important to design a simple workflow where statuses and transitions are carefully defined and their number is kept to a minimum, thus maintaining a simple and fluid work structure throughout the sequence of activities.

Transition number	Type of transition
1	Ticket creation
2, 13	Planning
3	Assignment
4	Start of activity
5, 9, 10, 12	Rejection
6	Pause
7	End of activity
8	Verified completion
11	Reopening
14	Retrospective – Closure

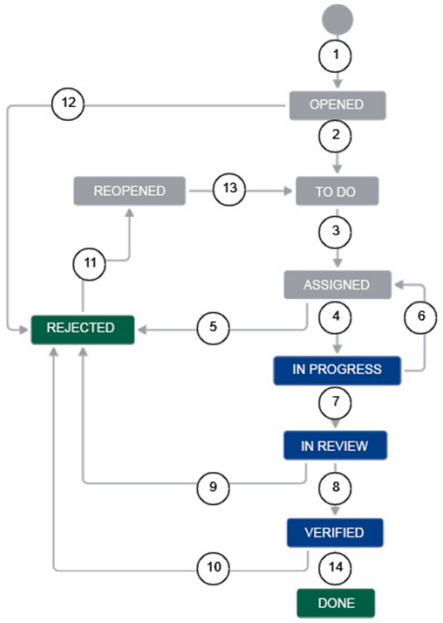

Figure 5 – Proposal of a ticket management workflow

The workflow proposed in Figure 5 uses the nine previously defined states (*opened, to do, assigned, in progress, in review, verified, reopened, rejected, done*), linked by ten different types of transitions (some types of transition appear several times in the workflow, bringing the total number of transitions to 14).

The transitions between statuses are defined as follows (see Figure 5):

① Ticket creation: when a ticket is created, it automatically transitions from an undefined status to the “opened” status. The set of created tickets constitutes the equivalent of the backlog defined in Scrum.

② Planning: similar to a Scrum team, the development team selects tickets to be handled over a given period of time and organizes their handling by assigning a priority to each ticket. As soon as the priority of the ticket is set, the operator changes its status to “to do”. The set of these “to do” tickets represents the sprint backlog. Transition 13 also corresponds to a planning action when a rejected ticket is reopened.

③ Assignment: as the development cycle unfolds, the development team selects the tickets to be handled from the sprint backlog. The assignment precedes the start of operations on a ticket and involves deciding who is responsible for processing the ticket.

④ Start of activity: as soon as the activity linked to a ticket starts, the ticket status changes from “assigned” to “in progress”. This can be done manually in some cases (e.g. writing of requirements), but it can also be done automatically at the creation of a development branch (e.g. in the case of a coding activity).

⑤ Rejection: a ticket can be rejected for various reasons, e.g. if an error occurs when editing the ticket. In all cases, the reason for rejecting a ticket must be specified in the ticket comments. Transitions 9, 10 and 12 are identical.

⑥ Pause: If, for any reason, the activities linked to the processing of a ticket are interrupted for a short period of time, the ticket must be returned to its “assigned” status so that the whole team is aware of the progress of the sprint.

⑦ End of activity: once the activity linked to the ticket has been completed, a person in charge of reviewing the work produced must be appointed (independence is made possible at this level). The status of the ticket is thus changed from “in progress” to “in review”.

⑧ Verified completion: the ticket transitions to the “verified” status once the review has been conducted and the product work validated, remaining there until the end of the development cycle (sprint).

⑪ Reopening: a rejected ticket can be reopened for editing.

⑭ Retrospective: a retrospective is conducted at the end of each development cycle. This operation entails reviewing all tickets handled during the sprint to verify one last time that all selected activities have been conducted as recommended, that all documents are present, and that traceability and other requirements have been met. Validating a ticket in retrospect is tantamount to deciding that the associated activity is completely finished and that there is no need to return to it thereafter. The ticket is thus assigned the “done” status, where it will remain until the end of the project (no action can be taken to change the status of a “done” ticket).

F. Ticket implementation and continuous development

The ticket system described earlier can be easily integrated into a continuous development process that iteratively strings the steps of a standard development cycle shown in

Figure 1.

To illustrate this point, Figure 6Figure 5 associates each ticket with the different phases of a DevOps-type software development cycle. Starting with the planning phase (“Plan”), the figure shows the preferred use by phase of certain ticket types for specifying, implementing and testing the expected software functionality. Tickets specific to the software development and verification phases (types 2, 3, 4, and 5) are associated with the “Create” and “Verify” activities. Tickets that concern the deployment of the application (types 6 and 9) can be associated with the “Package” and “Release” phases. Type 7 tickets correspond to the “Configure” phase, where the parameter files used by the software are defined. Lastly, ticket types 1 and 8 are incorporated into the “Monitor” and “Plan” phases to improve the processes and report any errors.

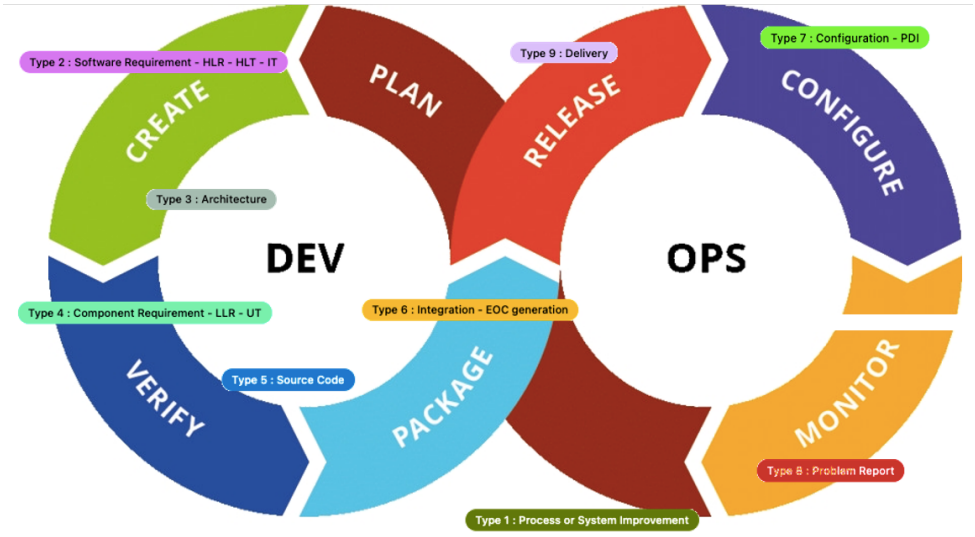

Figure 6 – Ticket implementation and continuous development

G. Independence, traceability and quality assurance

In the previous sections we have shown how a ticket-based management system can cover most certification requirements. Some requirements cannot be directly linked to a ticket type but can nevertheless be covered by the principles underlying the implementation of ticket-based management.
The following paragraphs explain how the principles of independence, traceability and quality assurance can be fulfilled using ticket-based management of software engineering data along with some complementary mechanisms.

Requirements from standards regarding independence, traceability and quality assurance have a significant impact on the work organization of the development teams. Depending on the level of quality assurance defined, independence between activities is relatively permissive. In general, development activities must be conducted independently of activities involving the writing of requirements and associated test cases. Similarly, only a person who has not contributed to the execution of a given activity should conduct its review.

As far as traceability is concerned, conventions must be established to identify the classification of requirements and the links between them. At the same time, tracing the link between an activity and its corresponding requirement is essential. In the context of software development, for example, the need for independence between the people implementing the source code and those in charge of writing the associated test cases requires a trace of the authors of these activities.

Quality assurance activities must be conducted throughout the engineering process and the associated evidence must be kept. Associating a checklist (to be followed during quality reviews) to each ticket type will help to define the scope of the review and to record date, author and observations of the quality review.

In our proposal, the regulatory requirements concerning traceability and independent reviews is supported natively through the use of tickets, as each ticket can be linked to any other one. Since the changelog of a particular ticket is kept, it is possible to retrieve information about the author of the ticket, its reviewer, other associated tickets, and the dates on which each change occurred. Table 7 lists the requirements that can be covered by the mechanisms underlying the use of tickets.

Table 7 – Reference framework requirements implemented through work organization and ticket-based management

Regulatory requirement	**Implementation using tickets**
[N1-SW11] The contractor shall ensure bidirectional traceability between the following elements: • Upstream requirements / software requirements • Software requirements / test cases • External interfaces / test cases • Test cases / test procedures • Test procedures / test results Note: traceability up to the code is not required for non-critical software.	Bidirectional traceability is ensured by the intrinsic nature of ticket-based management (links between tickets). Ticket-based management also provides additional traceability between the software requirements and the resulting code.
[N1-SW12] The contractor shall conduct reviews on the following elements: • Upstream software specification • Software specification • External interfaces • General design • Test cases • Test results	For its review, a ticket may be assigned to a member of a team directly related to the ticket. The assignment is decided when the ticket is issued, according to the rules of independence between the person creating the ticket and the person conducting the review.
[N1-SW16] The contractor shall conduct quality assurance activities to verify the application of plans (PDL and other internal plans if applicable) and rules throughout the project.	Some methods to ensure quality during development are specified in the tickets. As for the rest, they depend on the general organization of the project.

Ticket-based data management easily ensures the principles of traceability and independence. Links can be created between tickets and each action is logged to know what has been modified and by whom. The author of a ticket can automatically inform the reviewer that the activity has been completed, promptly triggering its review.

A potentially interesting approach to reinforce and systematize the review process is the "Pull/Merge Request" mechanism. At present, Pull/Merge Requests are mainly used at source code level to conduct peer reviews. The mechanism relies on reserving the right to publish on the common working directory (master) to a single experienced person. This person has the responsibility of commenting and validating the work of other developers before adding it to the shared working directory. The Pull/Merge Request thus corresponds to asking the owner of the master to consider the modifications that one wants to share. That way, the anarchic scenario where each developer can upload their work to the shared directory is not allowed. The prerequisite of having requested a peer review ensures that the work submitted complies with the expected quality and reliability criteria.

Our proposal seeks to generalize this approach to manage all software engineering data in digital format within a Git-like version management system. This approach offers the advantage of a complete baseline, managed in versions and with a complete history of development changes. It therefore seems appropriate to include the concept of Pull/Merge Request in our framework proposal for all phases of development, specification, design, source code implementation and associated tests. The goal is to reinforce the contribution of reviews by systematizing them on all the data produced at each iteration to develop a new functionality.

To summarize, our proposal enables to smoothly integrate certification into the software development process and removes all current biases that push it too late into the process. Our proposal also addresses certification constraints and allows for the continuous development of safety-critical software subject to certification. The last section of this article demonstrates the operational use of the previously described mechanisms by means of a tooling and a validation example of the framework.

IV. TOOLING AND VALIDATION OF THE FRAMEWORK

To implement the proposed framework, we first need to choose and configure a consistent set of support tools and make them work in concert, if we are to provide a structuring and functional framework for Agile documentation management [Gallina, 2015]. This section proposes an implementation of the framework and evaluates its effectiveness and efficiency on a case study. Lastly, it discusses our proposals.

A. Tooling choice and implementation

Several types of tools are required to implement the framework proposal: a ticket management tool; a source code version control system; an automation server for creating, deploying and automating the project; and document generation tools (e.g. software architecture descriptions in the form of diagrams).

Several alternatives are available on the market for each type of tool, such as SVN for the source code version control system or Jira for the management of tickets. There is little difference between all these widely used solutions; the advantage of GitHub is the ability to run our software for free in a virtual environment.

Table 8 – Overview of some candidate tools for supporting the framework proposal

Function	Stage of the framework	Available tools	Access	Usage
Ticket management	Digital management of engineering data	Jira	Freemium	Widely used
		GitLab	Open source	Widely used
		GitHub	Freemium	Widely used
		Azure DevOps	Freemium	Widely used
Source code version control system	File storage that includes the complete history of changes made to the file	Git	Open source	Widely used
		SVN	Open source	Widely used
		CVS	Open source	Moderately used
		Mercurial	Open source	Moderately used
Task automation	Automation of the build, test and deployment phases of software development, facilitating continuous integration and delivery	Jenkins	Open source	Widely used
		Bamboo	Freemium	Moderately used
		GitHub Actions	Freemium	Widely used
Architecture description	Definition of different digital architecture diagrams	PlantText	Open source	Moderately used
		PlantUML	Open source	Moderately used

From all the available tools, we have prioritized those that are easily accessible (open source, or free in their standard version), commonly used and interoperable, and that are thereby useful for building a truly consistent and operational framework.

For our tooling proposal, we have selected the following tools:

Git, an open-source decentralized version control tool, is our choice of source code version control system. We handle ticket management with the GitHub platform, which is dedicated to the project management of software development. Rather than dedicating a unique location for the software version history, each working copy of the code acts as a repository holding the complete change history. Git allows to 'commit'[7] new changes, create 'branches'[8], perform 'merges'[9], and compare old versions. In addition, GitHub offers the possibility to automatically run numerous tasks (tests, application deployment on the physical target, etc.).

In Figure 7, these tools are associated with the different phases of the framework.

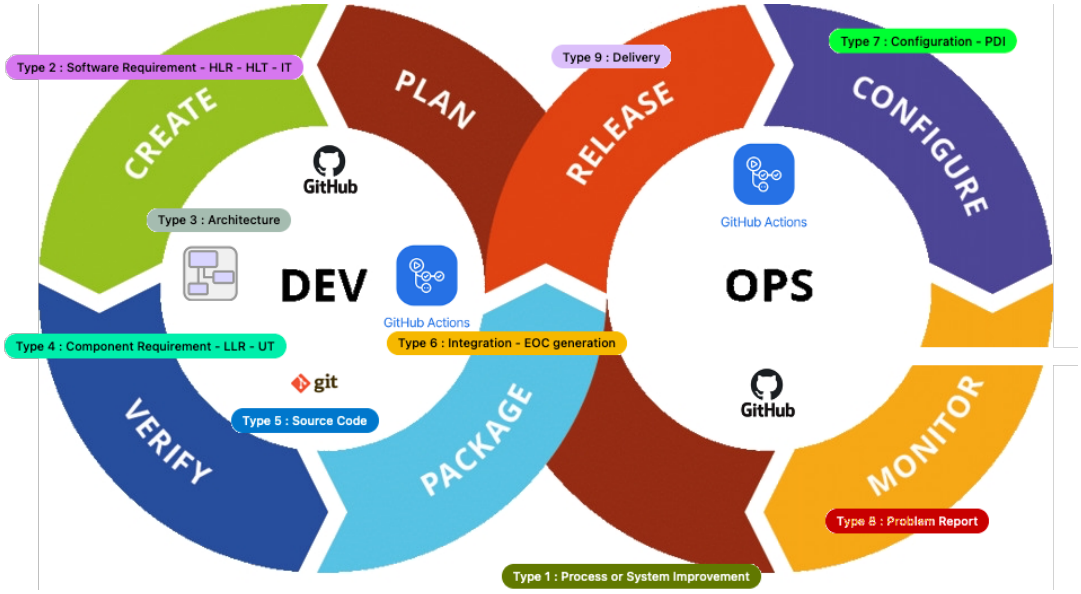

Figure 7 – Proposal of tool association to the different phases

Among other things, GitHub also supports the implementation of the Pull/Merge Request mechanism, reinforcing independence and review capabilities by recording review history, the changelog, and contributors to each ticket. With this mechanism, team members can notify other team members that they have completed an activity on their personal branch. Rather than arbitrarily contributing to the main branch shared with all project members, the author of the modification submits a Pull/Merge Request to another team member—often more experienced—so that the latter can validate the modification and merge it with the main branch. The Pull/Merge Request is more than a mere notification, however: it acts as a dedicated forum to discuss the proposed modification. Should the modification be problematic, team members can post comments to trigger a discussion and decide on necessary adaptations. The discussion log is kept.

The GitHub flow (Git workflow) describes in detail the logical steps of a Pull/Merge Request (see Figure 8):

- Create a branch to introduce a modification
- Create the commits associated with the requested modification

[7] A *commit* is a local record of a set of files and directories; it can be seen as a snapshot that captures the state of a project at a given time, which corresponds to a major step in the project timeline. Commits reflect a revision in the code; they include a date, an author, a textual description, and a link to previous commit(s). With the local repository of each developer acting as a buffer between their contributions and the remote centralized repository, the commit can then be "pushed" towards the latter. Thus, a commit constitutes a backup that can be used to return to a specific development stage at a later time.

[8] A *branch* works as an isolated workspace where various changes can be introduced, enabling several people to work on the same project without disrupting the development of the main code. The branch created by default when a repository is initialized is called the "master branch". Branches can be used separately or in collaboration, helping to structure and organize work on different project parts or at different stages of project maturity. A branch can then be simply dropped or merged with another branch.

[9] A *merge* allows to incorporate the work of a branch into another branch.

- Open a Pull/Merge Request
- Launch a discussion on the review of this modification
- Deploy and merge the modification with the main branch

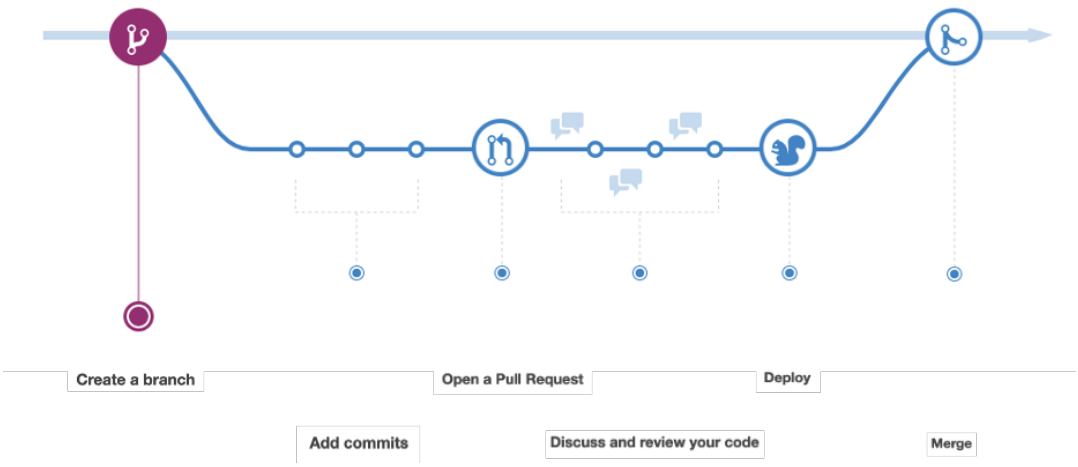

Figure 8 – Pull/Merge Request mechanism within a GitHub flow

We propose to apply the Pull/Merge Request mechanism to all engineering data, software specification, architecture, source code, unit tests, and integration tests, as depicted in Figure 9. We thus seek to exploit Pull/Merge Requests to systematize independent reviews of all engineering data while preserving the history of all changes applied to this data.

To that end, two separate teams are organized to address another independence requirement, i.e. between the team producing the specifications and the one in charge of the implementation:

- Production team: in charge of writing specifications in the form of tickets matching the features, as well as writing the associated tests.
- Development team: in charge of developing the new features.

The S1 specification shown in Figure 9 is described by a ticket generated by the production team. Once the ticket has been generated, an experienced engineer conducts a review. Then, two branches are created:

- B1 branch: the production team performs the test(s) associated (T1) with the S1 specification.
- B2 branch: the development team develops the function (F1) that corresponds to the intended functionality (associated with the S1 specification).

As soon as the writing of the tests is completed, the production team launches a Pull/Merge Request. The experienced engineer from the production team then receives a notification:

- If the tests comply with the specification, the engineer merges the B1 branch with the master branch.
- Otherwise, the engineer does not authorize the merge and discusses the detected noncompliance with the production team.

The process is the same for the implementation of source code functionality: as soon as the development of the F1 function is completed, the development team launches a Pull/Merge Request. The experienced engineer from the production team then receives a notification:

- If the function complies with the specification, the engineer merges the B2 branch with the master branch.
- Otherwise, the engineer does not authorize the merge and discusses the detected noncompliance with the development team.

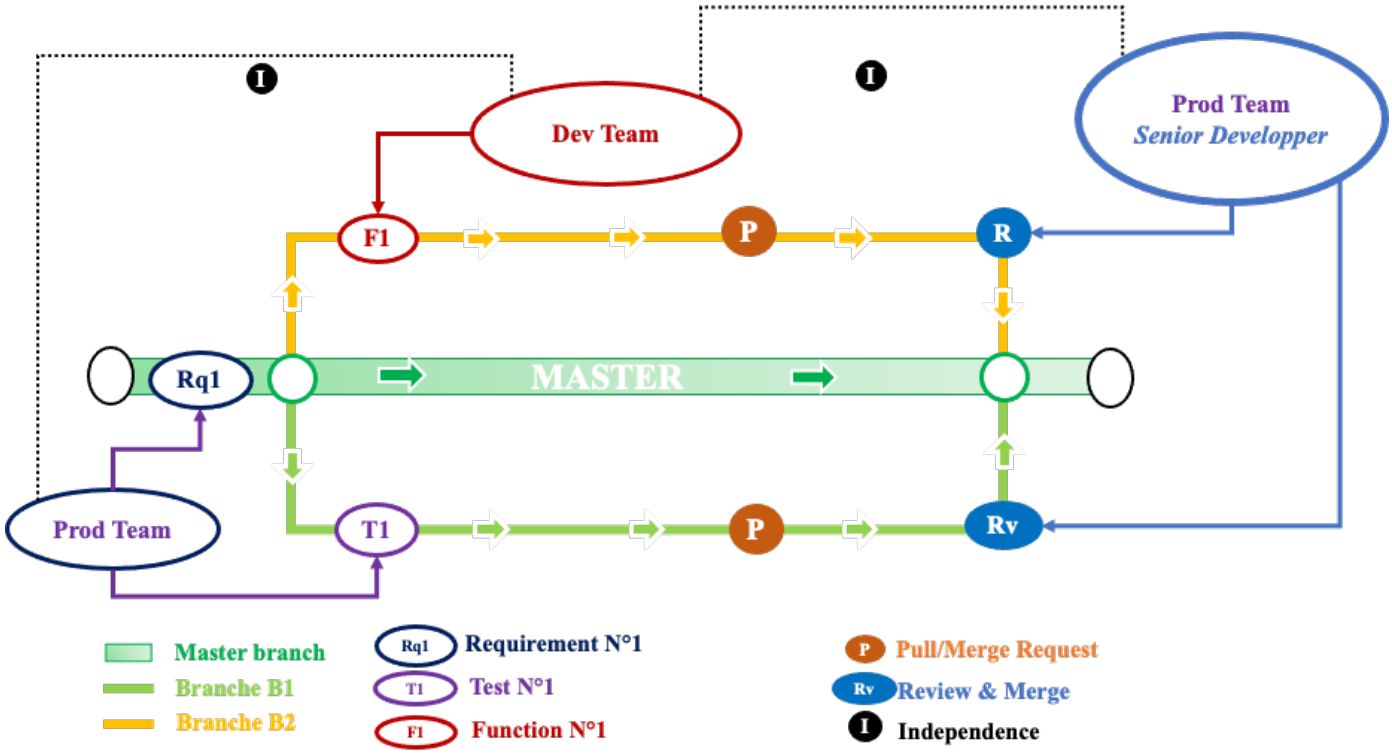

Figure 9 – Use of the Pull/Merge Request mechanism in our proposal

It is worth noting that, even though we chose GitHub for the tooling proposal of this article, our framework could just as easily have been implemented with different tools (including those listed in Table 8). For example, other successful tooling experiments have been conducted using Azure DevOps, Bitbucket and Jira.

B. *Case study*

Our tooling framework was used in several case studies. Here, we demonstrate its application in the development of the Linty software tool (https://sonar.linty-services.com), which analyzes the quality and reliability of VHDL code by verifying its compliance with coding rules. Linty offers several features to ensure that the VHDL code developed to program FPGA electronic components complies with recognized standards, such as the CNES VHDL Handbook (https://github.com/VHDLTool/VHDL_Handbook_CNE) or the DO-254 User Group (Best Practice VHDL Coding Standards for DO-254 Programs, Rev 1 of February 26, 2021). The development of Linty requires high levels of quality and reliability, for the tool to be used natively in FPGA security development (such as DAL A). If the verification of coding rules is conducted by a tool for designing a critical component, its correct functioning must be demonstrated. A qualification kit must be built to ensure that each feature performs properly in the user environment. Consequently, all coding rules—new or old—are specified, tested, integrated, monitored and deployed in a fully automated manner to validate the proper execution of the rule. Once all these tests have been completed, some are delivered to the end users for them to execute in their operational environment and thus validate the expected behavior.

In this section, we will explain how the framework proposal is applied to introduce a new coding rule to the VHDL language, using the principles previously explained.

1) *Application of the framework*

Linty software has an incremental design. According to specific user needs, the development team selects the rules to implement one by one, progressively building all the engineering data (specification, unit tests and integration tests) required to verify its proper functioning. The production process of the new rule relies on the framework proposal. Since the high-level software specification and architecture of Linty have already been established, this example will focus on the description of a new coding rule, to be formalized in the form of a detailed LLR to explain

the behavior expected by the introduction of this new rule. In our framework, this description corresponds to a type 4 ticket (as defined in Table 5).

Figure 10 – Type 4 ticket of the detailed VHDL coding rule shows the creation of a type 4 ticket; the header specifies:

- the rule to implement (*Signal initialized on declaration requirement*)
- its author (*Vincent-Louis-DGA*)
- the ticket number (*#13*)
- its status (*Open*)
- the creation date (*12 April*)

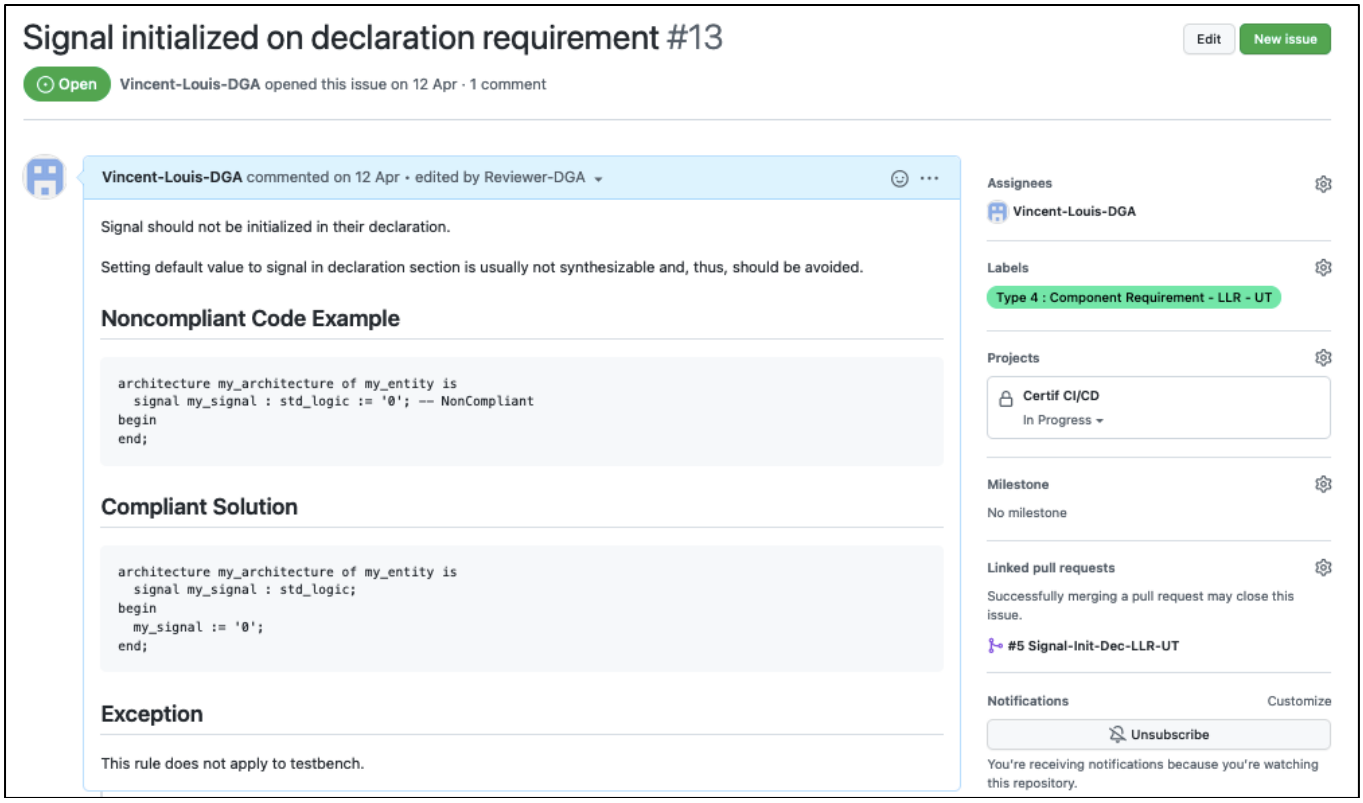

Figure 10 – Type 4 ticket of the detailed VHDL coding rule

When the team in charge of development pushes the ticket into the "in progress" status (according to the workflow in Figure 5), a detailed description of the specification and associated unit tests must be included.

The unit tests are often expressed as a set of source code files that can be linked to this ticket through the Linked Pull/Merge Request field (bottom right of

Figure 10 – Type 4 ticket of the detailed VHDL coding rule). In this example, a specific Pull/Merge Request *#5 Signal-Init-Dec-LLR-UT* was created to add the unit test files to the main project branch. While the Pull/Merge Request lists all the modified files, GitHub provides all the mechanisms to conduct the review of each of these files before performing the merge.

Figure 11 shows an extract of the seven files that were modified or created. Some of them concern the implementation of the tests and others the VHDL code files on which the tests will be executed to validate the behavior. The added files are shown in green. Two had already been reviewed (*VhdlLanguageRulesDefinitionTest.java*, *CheckListTest.java*), while the test file

SignalInitializedOnDeclarationBetaCheckTest.java is presented in detail for review. The latter describes the unit tests to be executed to ensure that the new coding rule is well implemented.

Four other files correspond to the VHDL test files, where the test must be executed to determine if the behavior is as expected.

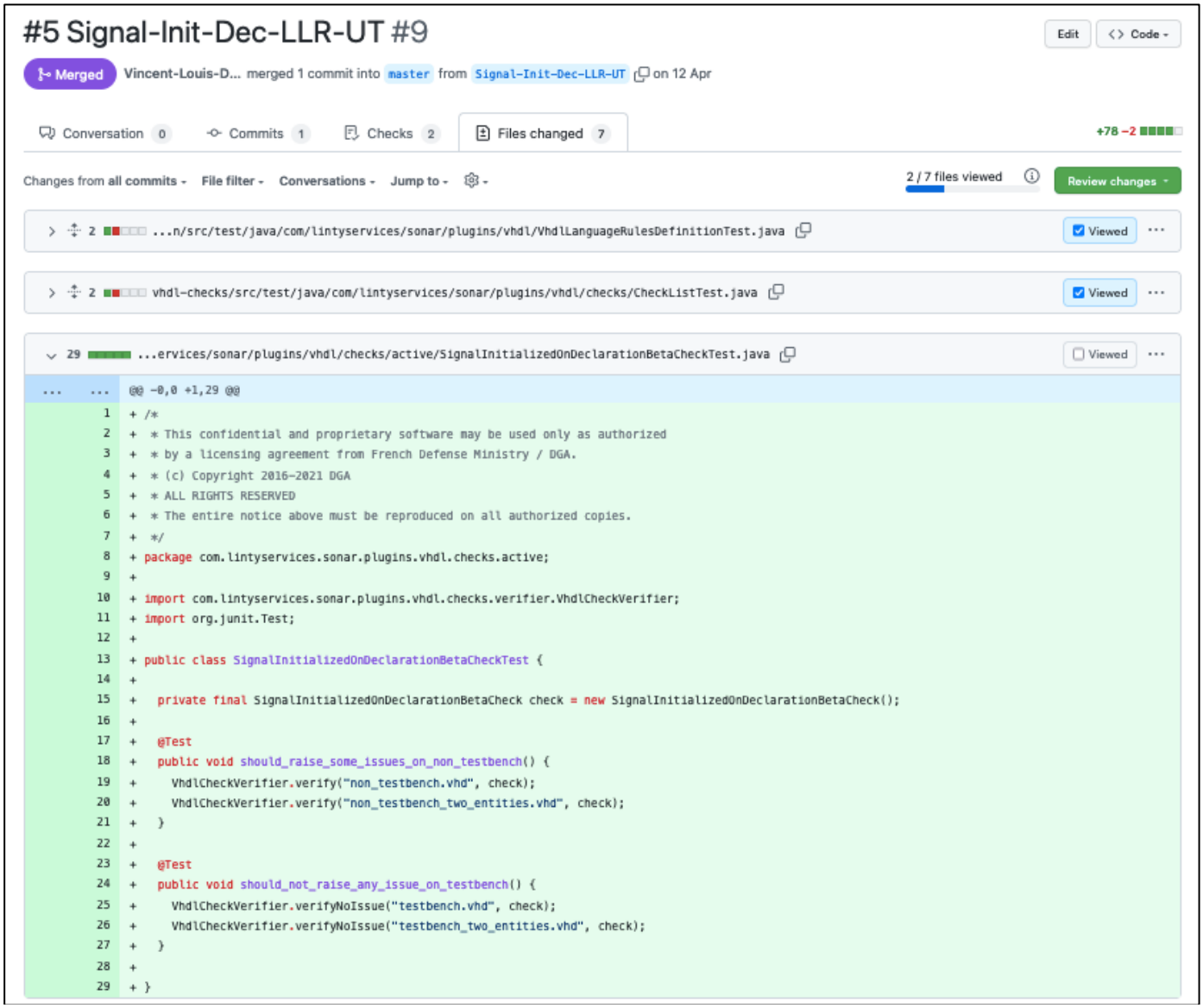

Figure 11 – Pull/Merge Request associated with type 4 ticket #13

Several test files are created to indicate where should a deviation from the coding rule be detected. Various scenarios are foreseen to deal with different cases of robustness (exception for testbench files, files with multiple entities, etc.). All these tests are fully automated and can be systematically rerun when a new feature is added to ensure the lack of regression.

Similarly, a type 5 ticket (“Source code”) was created to add the executable code that enables the implementation of the new rule. In relation to this type 5 ticket, Figures 12 and 13 show the four files that were specifically created: one to activate the rule in the quality profile proposed by default (*VhdlProfile.java*) and three others describing the algorithm that enables to detect the problem by correctly reporting it in the user interface.

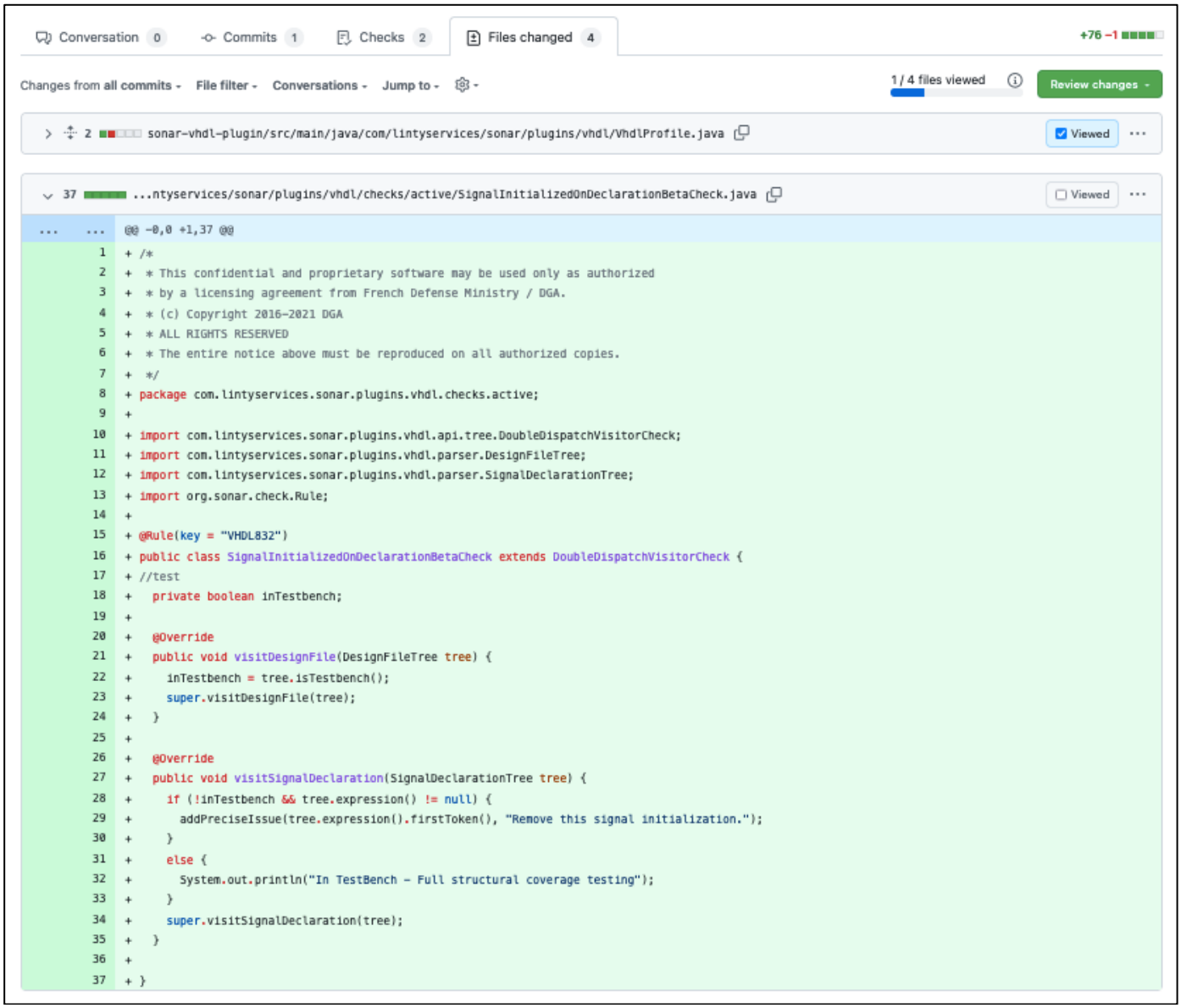

Figure 12 – Implementation of the coding rule "Signal initialized on declaration requirement"

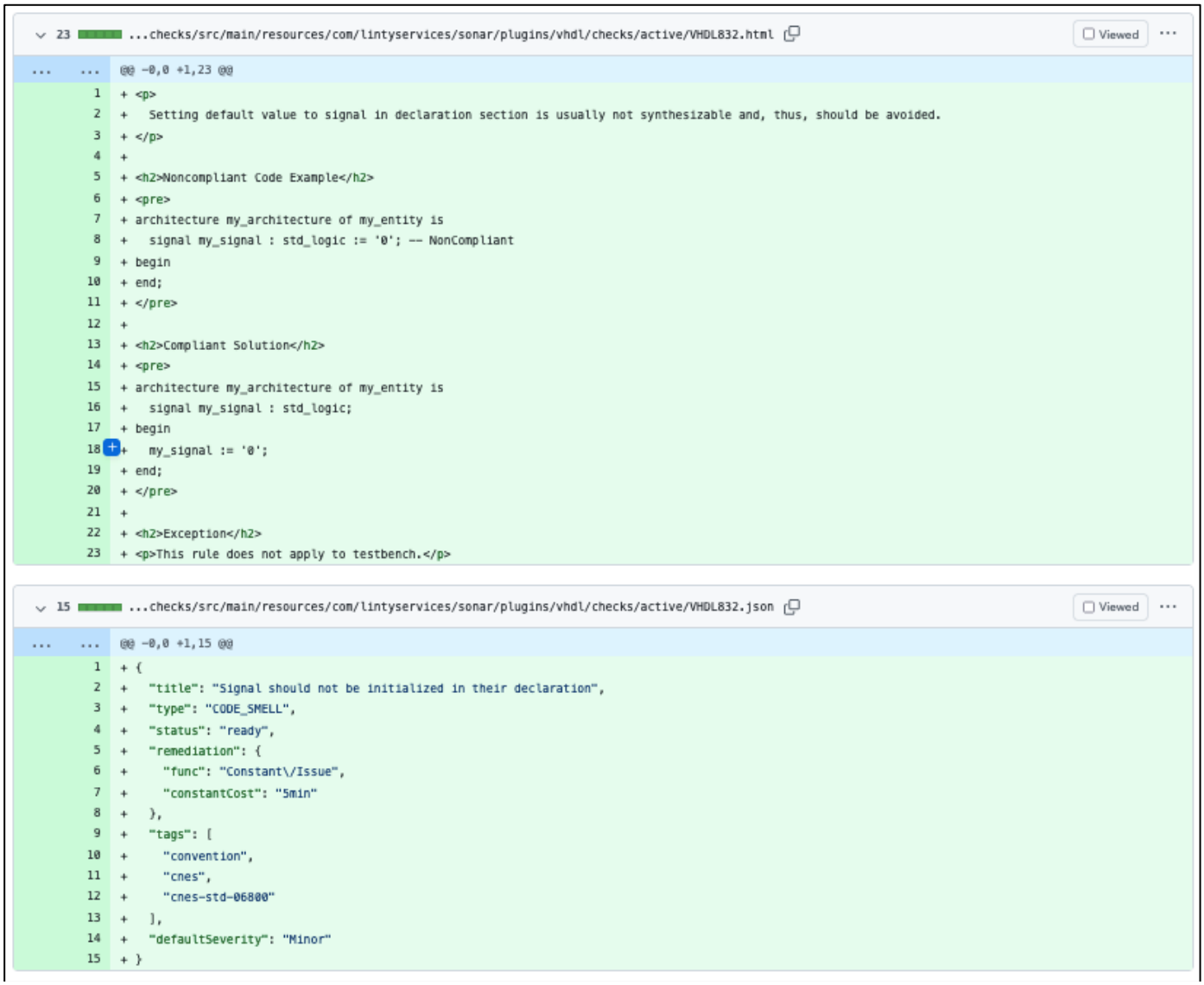

Figure 13 – Technical description of the rule that will be presented to users upon error detection

Finally, some integration test files were created to ensure that the errors (which the new rule is supposed to identify) will be detected when Linty is executed in its target environment.

2) Task automation

Once the specification, implementation and testing activities were completed, a configuration file for task automation was created. To that end, we used the GitHub Actions tool — a new service that directly automates the software development life cycle on GitHub using event-driven triggers. These triggers are specified events (e.g., Pull/Merge Requests). Every automation in GitHub Actions is handled using workflows (i.e. YAML files placed in the *.github/workflows* directory of a GitHub repository), which define the automated processes to be executed and enable the implementation of DevOps principles according to the sequence of phases shown in Figure 7.

In the example shown in Figure 15, our goal is to automate the various tasks at each phase in accordance with the general principles described in Figure 14, which depicts the DevOps cycle implementing the principle of continuous development: program the software in IDEs (Eclipse, Visual Studio); version manage the source code (Git, GitLab, SVN); build the application (Maven, Jenkins); verify the quality (SonarQube); test the behavior (JUnit); package the set of dependencies (Nexus, Artifactory); create an image (Docker); deploy to target and monitor the operation (Kubernetes).

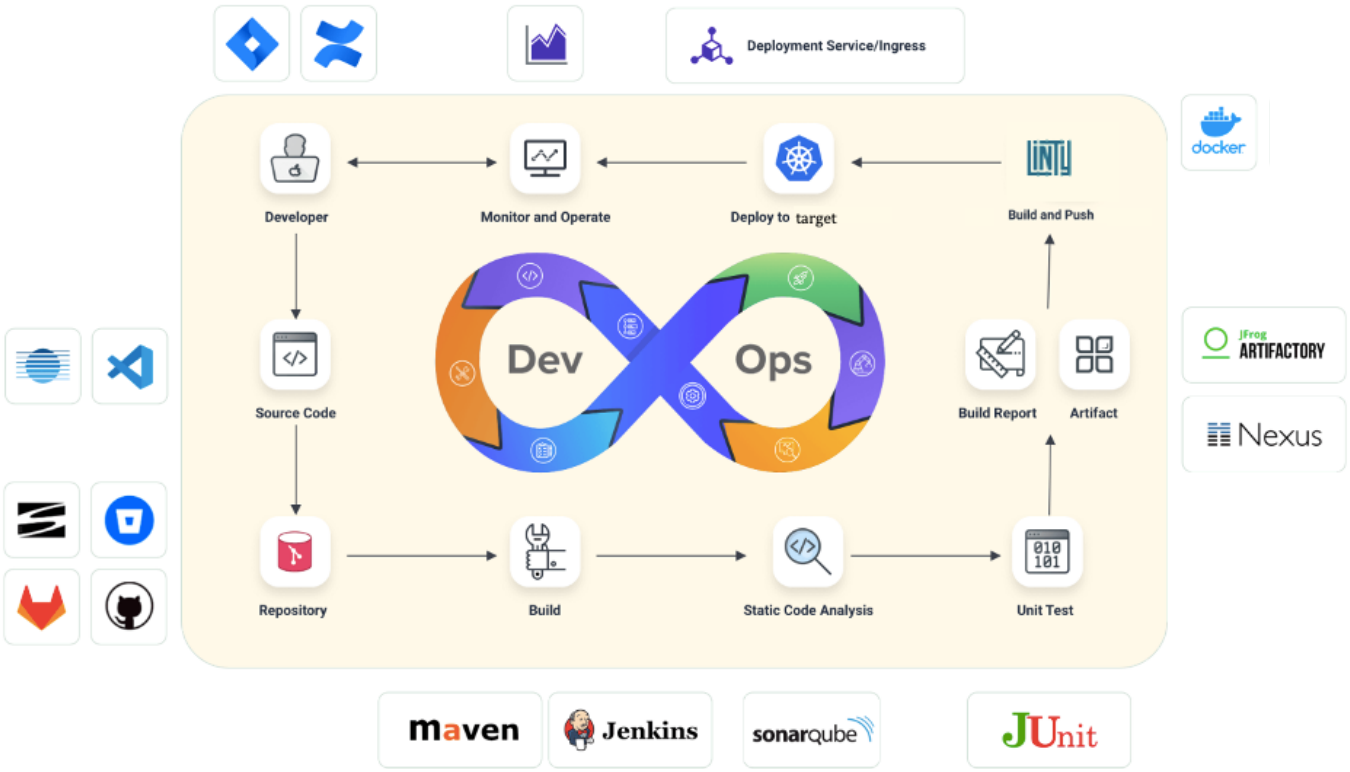

Figure 14 – General principles of the phases that can be automated using GitHub Actions

Figure 15 illustrates the specific instantiation of these general principles in our case study, describing the various steps as a log:

- Extract the source code from the Git repository (*Git Checkout*)
- Build the Linty plugin and run both unit and integration tests (*Build and execute UT/IT*)
- Load the plugin onto a virtual Ubuntu instance (*Upload Plugin JAR*)
- Analyze the quality of the Java code written to implement the rule (*Run SonarQube Analysis*)
- Load the plugin onto the target (*Download Plugin Jar and copyfile via ssh passwords*)

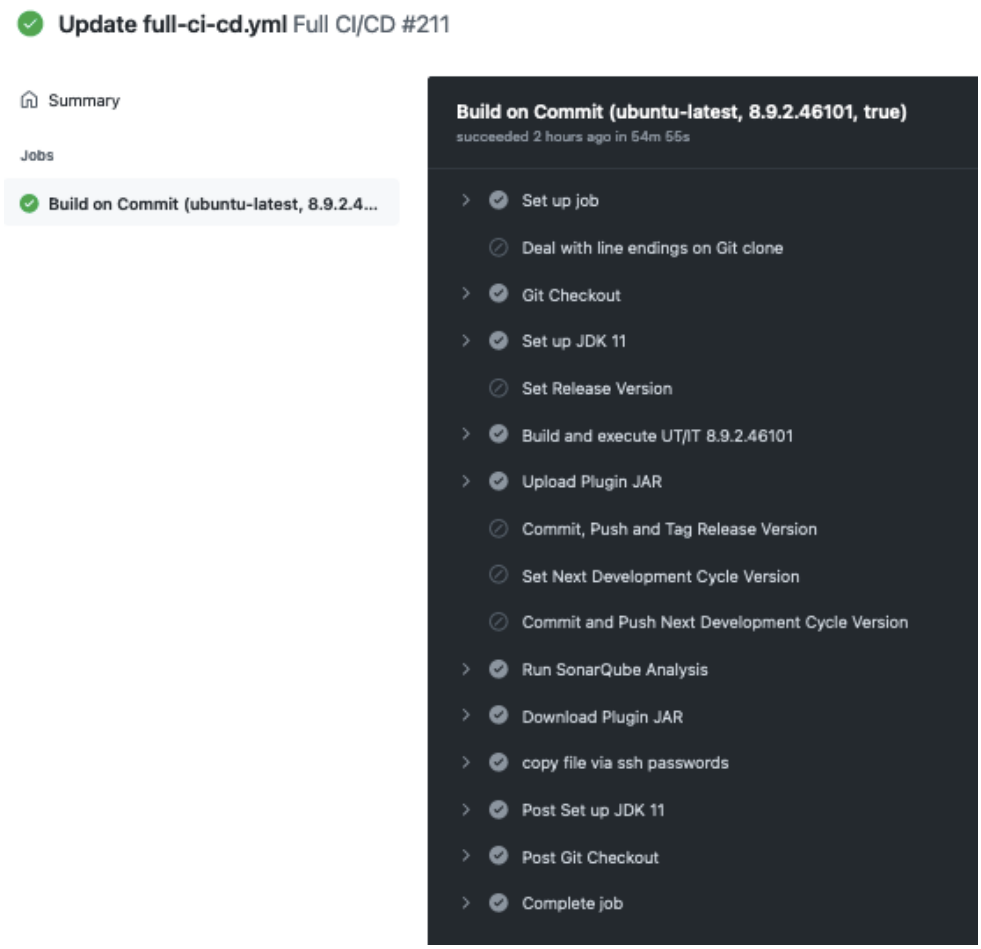

Figure 15 – Log of tasks automatically executed by GitHub Actions

The purpose of these automatic tasks—triggered at each merge on the master branch—is to build the Linty application, to reconduct all the unit tests (including those newly added), to deploy Linty in a virtual environment to ensure that the integration tests run smoothly, and to measure the quality of the code developed with the SonarQube static code analysis tool.

As shown in Figure 16, SonarQube is an open-source platform that provides a concise view of the quality and reliability of the code by pooling the results of the verification of coding rules, the duplicated code ratio, and the structural coverage rate of the code obtained from running the tests.

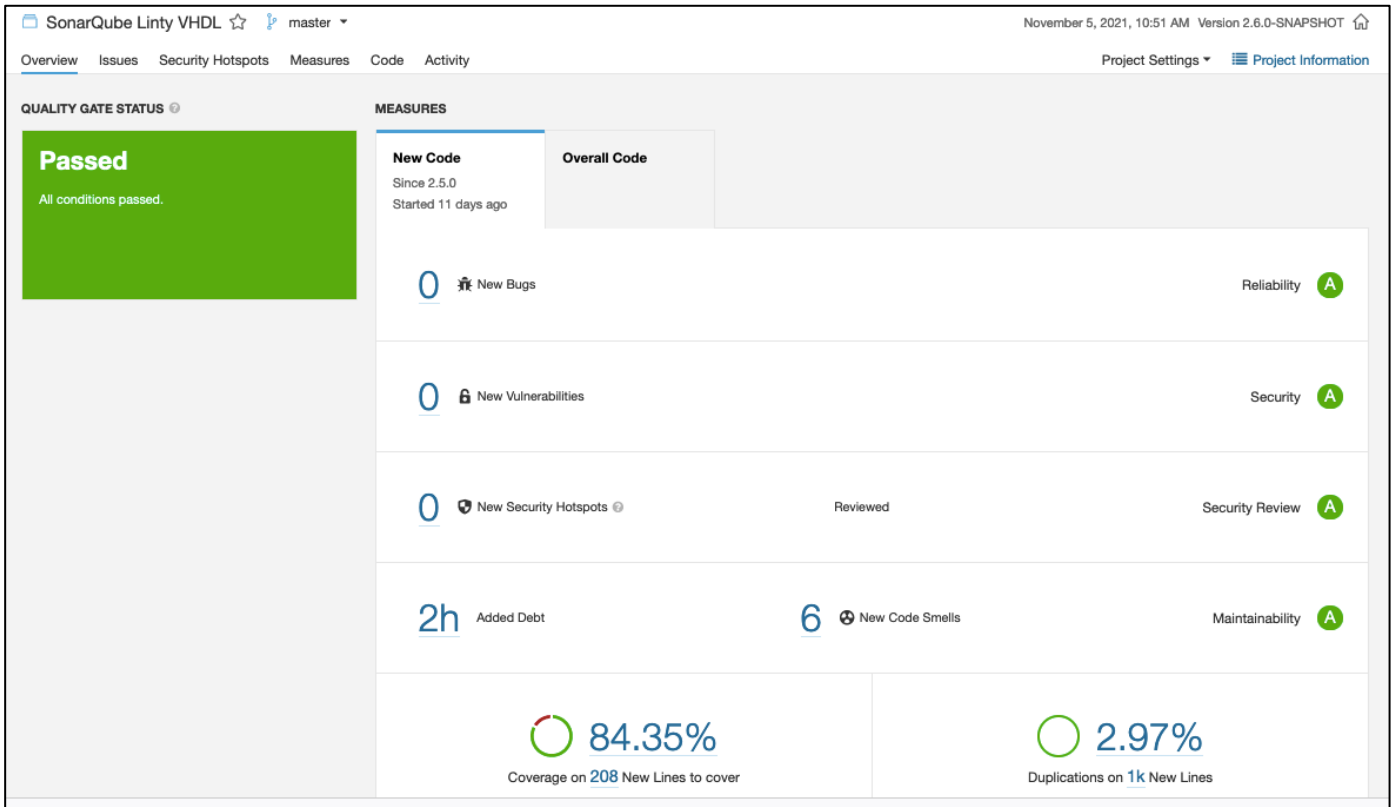

Figure 16 – Example: result of a SonarQube analysis of the quality of Java code written to implement the coding rule "Signal initialized on declaration requirement"

Provided that the automated process in GitHub Actions is successful, the last phase consists in automatically deploying the Linty plugin on a server accessible to all users (https://sonar.linty-services.com). As new coding rules are added, they are displayed in the VHDL quality profile available at:

https://sonar.linty-services.com/profiles/show?language=vhdl&name=VHDL+default

The whole automated procedure takes approximately ten minutes to complete. Should one of the steps not work properly, the prompt feedback allows to correct the problem immediately.

While this article describes the application of the framework to a type 4 ticket, the process is similar for other ticket types. Thus, by following our framework proposal, the Linty plugin will continuously improve. At the time of writing, over a hundred VHDL coding rules were already available, covering a wide range of good coding practices proposed by the CNES and the User Group DO254, whose coverage matrix is available at: https://linty-services.atlassian.net/wiki/spaces/STANDARDS/overview. In addition, what we have demonstrated can be generalized to other scenarios, regardless of the context (case study or not) and choice of tools.

C. Discussion and areas for improvement

The feasibility of our framework was demonstrated and validated by an adapted tooling proposal based on popular off-the-shelf solutions, most of which are open source. The ticket-based management approach helped us understand how to ensure traceability between all engineering data, as the method automatically records a complete changelog.

Proof of independent reviews is readily available by means of the Pull/Merge Request, which simplifies branch management and review of data production, thus ensuring that the people in charge of producing the specification and tests are not the same as those who implemented the source code.

All repetitive tasks are automated by a technical solution that is fully integrated into the software production process (application compilation, test execution, analysis of code quality and deployment to application target). Task automation allows for the entire software to be compiled, 300 existing tests to be run and the code to be analyzed for compliance (more than 400 coding rules for Java) in few minutes.

As a result, the delivery time for each new feature is minimized. No more than half a day is required to add a new, complex VHDL coding rule in full compliance with the system. When a simple rule or a rule similar to an existing one is added, less than an hour is required to complete the entire workflow; for a new feature, the production process takes only half a day (for two people to ensure independence between reviews). The team is immediately informed of any problem, which can be quickly corrected before the new version is redeployed.

Applying the framework to a case study helped to evaluate its effectiveness (through the quality of the results) and its efficiency (by assessing the effort required to adapt the framework and its intuitive implementation). Our framework meets the essential requirements for the certification of safety-critical software. However, research is still ongoing, and the DGA launched a study among various companies working in the defense field to statistically study which engineering activities eliminate the most development errors. The objective of this study is to have figures to select the priority constraints to be imposed in the development of critical software for defense systems. It aims to demonstrate that the generalization of independent reviews performed on all engineering data at the time of their production brings a lot of value in the context of a fluid production flow.

We can consider a few areas for improvement, in particular the execution time of the automated process, which currently stands at approximately ten minutes for more than 300 tests. Using more powerful servers would save valuable time to streamline the operational use of the service by the development team. Problem detection—including noncompliance with coding rules or insufficient coverage—must be accomplished as quickly as possible, so that developers can improve their production in near real-time. Obtaining these indicators as soon as possible is therefore crucial, e.g., when the source code file is saved for compliance with coding rules, at the level of the Pull/Merge Request for the coverage rate. We could also envision transposing these review principles to all engineering documents of the system (risk analyses in particular) so that the deployment authorization for any new major release be based on a consistent set of documents. TuSimple—a U.S.-based startup company that seeks to deploy a fleet of fully autonomous vehicles for road freight transportation—has already integrated this principle into its approach [TuSimple, 2021].

V. Conclusion

This article proposes a framework and a tooling to deploy continuous certification, demonstrating in a case study that the mechanisms used to create all the engineering data (specification, code, unit tests and integration tests) comply with the requirements of the certification process. This ensures traceability between all elements, providing evidence that they have all been independently reviewed and that some activities are conducted by different individuals.

Our framework proposal is hybrid in nature, combining the principles of a variety of certification-compatible frameworks: frequent releases and automation of repetitive tasks (tests in particular) found in DevOps and Scrum; the rituals (Daily Meeting, Planning Poker), notion of “Minimal Valuable Product”, and distribution of roles among team members specific to Scrum; and the Pull/Merge Request mechanism on all software engineering data enabling the systematization of independent reviews.

The industrialization of this approach, tested by a team in charge of developing an application that runs on a safety-critical embedded computer in aeronautics, suggests a much higher productivity than those announced by the few other operational initiatives of its kind. The major challenge lies in the paradigm shift—which includes methods and tools—that it represents for players who have already implemented complex processes to comply with certification requirements. The associated initial investment and the fear of risk-taking during a certification audit can be major obstacles to the deployment of this approach. Trained and competent resources are also required for these breakthrough solutions, but are still rarely taught in engineering curricula.

REFERENCES

[ARP4754A 2011] Society of Automotive Engineers, Aerospace Recommended Practice “Guidelines For Development Of Civil Aircraft and Systems”, November 2011.

[ARP4761 1996] Society of Automotive Engineers, Aerospace Recommended Practice “Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment”, December 1996, https://www.sae.org/standards/content/arp4761/

[Baron 2021] Baron C., Louis V., " Towards a continuous certification of safety-critical avionics software", Computers in Industry, vol. 125, février 2021. https://doi.org/10.1016/j.compind.2020.103382

[Beck 2001] Beck, K. et al., " The Agile Manifesto", Agile Alliance, 2001, Retrieved March 2019.

[Balaji 2012] Balaji, S. and M. Sundararajan Murugaiyan. “WATEERFALLVs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC”, 2012.

[Chenu 2013] Chenu E., "Integration Continue”, Séminaire Ingénierie des Systèmes Complexes à Logiciels Prépondérants, ISCLP, Toulouse 2013.

[Chenu 2022] Chenu E., "Gestion de configuration et intégration continue de versions parallèles et cohérentes”, Séminaire Ingénierie des Systèmes Complexes à Logiciels Prépondérants, ISCLP, Toulouse 2022.

[DoD 2018] DoD, Defence Science Board, “Design and acquisition of software for defense systems, February 2018, https://dsb.cto.mil/reports/2010s/DSB_SWA_Report_FINALdelivered2-21-2018.pdf

[EASA CS-25 2018] EASA, “Certification Specifications for large aeroplanes, Amendment 21”. March 2018. https://www.easa.europa.eu/sites/default/files/dfu/CS-25%20Amendment%2021.pdf

[Edeki, 2015] Edeki, C. “Agile Software Development Methodology”. European Journal of Mathematics and Computer Science, 2, 2015.

[Fowler 2010] Fowler M., “Continuous Integration”. ThoughtWorks, May-2010. [Online]. Available: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf

[Gallina 2015] Gallina Barbara, Nyberg Mattias, “Reconciling the ISO 26262-compliant and the agile documentation management in the Swedish context”, Proceedings of the third Workshop on Critical Automotive applications: Robustness & Safety (CARS), joint event of EDCC-2015, Paris, France, September 8th, 2015.

[Gallina 2018] Barbara Gallina, Faiz Ul Muram, Julieth Patricia Castellanos Ardila, “Compliance of Agilized (Software) Development Processes with Safety Standards: a Vision”, 4th international workshop on agile development of safety-critical software (ASCS), May 21st, Porto, Portugal, 2018.

[Gaudin 2013] Gaudin O., “Continuous Inspection - A Paradigm Shift in Software Quality Management”, SonarSource, 2013.
Available: https://www.sonarsource.com/docs/sonarsource_continuous_inspection_white_paper.pdf. [Accessed: 13-Oct-2021].

[Hilderman 2009] Hilderman Vince, “DO-178B Costs Versus Benefits”, HighRely White Paper, 2009. http://www.highrely.com/whitepapers.php

[Humble 2010] Humble J., Farley D., “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, Addison-Wesley, 2010, ISBN-10 : 9780321601919

[IEC 61508 2010] International Electrotechnical Commission, “Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems”, https://www.iec.ch/functionalsafety/standards/

[Kuehne2020] Uwe Kuehne - Airbus Defence and Space, Germany, “Introducing Agile Methodology into Advanced Systems Engineering Training”

[Kumar and Bhatia, 2012] Gaurav Kumar, P. Bhatia, “Impact of Agile Methodology on Software Development Process”, 2012.

[LaPlante 2018] LaPlante W., Wisnieff R., “Design and Acquisition of Software for Defense Systems”, 2018. https://dsb.cto.mil/reports/2010s/DSB_SWA_Report_FINALdelivered2-21-2018.pdf

[Laprie 1996] Laprie J-C., “Guide de la sûreté de fonctionnement”, Cépaduès, 1996.

[Lemoussu 2018] Lemoussu S., Chaudemar J.-C., Vingerhoeds R.A., “Systems Engineering and Project Management Process Modeling in the Aeronautics Context: The SMEs Study Case”, International Journal of Mechanical and Mechatronics Engineering, vol. 12 (n° 2). pp. 88-96, 2018.

[Leveson 2003] Leveson, Nancy. “White Paper on Approaches to Safety Engineering”, April 2003. http:// sunnyday.mit.edu/caib/concepts.pdf

[Louis 2019] Louis V., Baron C., " Vers une certification continue des logiciels critiques en aéronautique ", Techniques de l’Ingénieur, 27 p, novembre 2019.

[Marsden 2018] Marsden J., Windisch A, Villermin J., Aventini C., Mayo R., Grossi J., Fabre L., ”ED-12C/DO-178C vs. Agile Manifesto – A Solution to Agile Development of Certifiable Avionics Systems”, Conférence Embedded Real Time Software and Systems (ERTS2), Toulouse, France, February 2018.

[Mrabti 2018] Mrabti A., Gautherot D., Brossard V., Moy Y., Pothon F., “Safe and Secure Autopilot Software for Drones”, Conférence Embedded Real Time Software and Systems (ERTS2), Toulouse, France, February 2018.

[Ninni 2021] Ninni L., Blog Launizo consulting, https://www.launizo.com/blog/methodes-et-outils-de-productivite-en-entreprise-1/post/les-methodes-agiles-3 - consulted October 2021.

[NT DGATA 2016] DGA Techniques aéronautiques, Note Technique 16-DGATA-P1301261003001-1P-C “Référentiel d’exigences d'ingénierie des logiciels et composants électroniques complexes pour la prise en compte de la sûreté de fonctionnement”, 2016.

[Rahman, 2015] A. A. U. Rahman, E. Helms, L. Williams and C. Parnin, "Synthesizing Continuous Deployment Practices Used in Software Development", Agile Conference, , pp. 1-10, 2015.

[Rempel 2014] Rempel, Patrick; Mäder, Patrick; Kuschke, Tobias; Cleland-Huang, Jane. “Mind the Gap: Assessing the Conformance of Software Traceability to Relevant Guidelines”, International Conference on Software Engineering (ICSE), New York, USA, ACM: 943–954, 2014.

[RNC-ECSS-Q-ST-80 2017] European Cooperation for Space Standardization ECSS-Q-ST-80C Rev.1 – Software product assurance, February 2017.

[RTCA DO-178C 2012] RTCA SC-205, EUROCAE WG-12, DO-178C/ED12C, “Software Considerations in Airborne Systems and Equipment Certification”, January 2012.

[RTCA DO-254 2006] RTCA and EUROCAE, RTCA DO-254/EUROCAE ED-80 “Design assurance guidance for airborne electronic hardware”, 2006.

[Safe 2021] Scaled Agile, “System Team”, updated 10/02/2021. https://www.scaledagileframework.com/system-team/

[Scrum 2018] Scrum.org, “What is Scrum?”, consulted 02/12/2018. https://www.scrum.org/resources/what-is-scrum?

[Steghofer 2019] Jan-Philipp Steghöfer, Eric Knauss, Jennifer Horkoff, Rebekka Wohlrab, “Challenges of Scaled Agile for Safety-Critical Systems”, 2019.

[Stellman and Greene, 2013] Andrew Stellman, Jennifer Greene, “Learning Agile: Understanding Scrum, XP, Lean, and Kanban”, 2013.

[Thummadi et al., 2011] B. Veeresh Thummadi, Omri Shiv, Nicholas Berente, Kalle Lyytinen, “Enacted Software Development Routines Based on Waterfall and Agile Software Methods: Socio-Technical Event Sequence Study”, Service-Oriented Perspectives in Design Science Research - 6th International Conference, DESRIST 2011, Milwaukee, WI, USA, May 5-6, 2011, Lecture Notes in Computer Science book series (LNCS, volume 6629).

[TuSimple 2021] AI Houry, “TuSimple’s Driver-Out Pilot Safety Framework”, December 2021, https://www.tusimple.com/wp-content/uploads/2021/12/TuSimple_Driver_Out_Pilot_Safety_Framework_Executive_Summary.pdf.

[Vöst 2016] Vöst S., Wagner S., “Towards Continuous Integration and Continuous Delivery in the Automotive Industry”, 2016.