

Highlights

Guidelines for the development of a critical software under emergency

Andrea Bombarda, Silvia Bonfanti, Cristiano Galbiati, Angelo Gargantini, Patrizio Pelliccione, Elvinia Riccobene, Masayuki Wada

- Developing safety-critical systems under emergency, e.g. with tight time constraints and by establishing a heterogeneous team composed of volunteers, requires both rigorousness and agility.
- This work offers lessons learned and guidelines that have been extracted from the experience of the development of a mechanical ventilator for COVID-19.

Guidelines for the development of a critical software under emergency

Andrea Bombarda^a, Silvia Bonfanti^a, Cristiano Galbiati^{b,c,d}, Angelo Gargantini^a, Patrizio Pelliccione^c, Elvinia Riccobene^e, Masayuki Wada^f

^aDepartment of Management, Information and Production Engineering, University of Bergamo, Bergamo, Italy

^bPrinceton University, Princeton, NJ, USA

^cGran Sasso Science Institute (GSSI), L'Aquila, Italy

^dINFN Laboratori Nazionali del Gran Sasso, L'Aquila, Italy

^eUniversity of Milan, Milano, Italy

^fAstroCeNT, N. Copernicus Astronomical Center, Polish Academy of Sciences, Warsaw, Poland

Abstract

Context. During the first wave of the COVID-19 pandemic, an international and heterogeneous team of scientists collaborated on a social project to produce a mechanical ventilator for intensive care units (MVM). MVM has been conceived to be produced and used also in poor countries: it is open-source, no patents, cheap, and can be produced with materials that are easy to retrieve.

Objective. The objective of this work is to extract from the experience of the MVM development and software certification a set of lessons learned and then guidelines that can help developers to produce safety-critical devices in similar emergency situations.

Method. We conducted a case study. We had full access to source code, comments on code, change requests, test reports, every deliverable (60 in total) produced for the software certification (safety concepts, requirements specifications, architecture and design, testing activities, etc.), notes, whiteboard sketches, emails, etc. We validated both lessons learned and guidelines with experts.

Findings. We contribute a set of validated lessons learned and a set of validated guidelines, together with a discussion of benefits and risks of each guideline.

Conclusion. In this work we share our experience in certifying software for healthcare devices produced under emergency, i.e. with strict and pressing time constraints and with the difficulty of establishing a heterogeneous development team made of volunteers. We believe that the guidelines will help engineers during the development of critical software under emergency.

Keywords: Safety-critical systems development, Software certification, Lessons learned, Guidelines, Healthcare

1. Introduction

Certification is a mandatory step [1, 2, 3] for the development of safety-critical devices, since a software failure or malfunctioning can compromise the health of human beings that interact with it. Safety-critical systems certification is required in many domains, including avionics [4, 5, 6], robotic applications [7], cyber-physical systems [8], and healthcare [9].

In healthcare, a software certification is required when the software is deployed on a medical device and controls it [10, 11, 12, 13, 14]; the software certification is regulated by the FDA Guidelines [15] and the IEC 62304 [16]. IEC 62304 defines safety classes of the software, based on the potential to create an injury to the patient: Class A – no injury or damage to health is possible, Class B – non-serious injury is possible, and Class C – death or serious injury is possible. The standard itself prescribes a set of activities, which must be performed (and documented) during the software development process.

Regardless of the class the software belongs to, the standard requires a software development plan, as well as software re-

quirements specification documents. A software development plan establishes the process to be followed during the development, the expected deliverables, and the software development life cycle model. Once the software requirements are defined, the software architecture design is derived for software belonging to class B or class C. Furthermore, a software detailed design for each software unit is required for class C. System testing is the only required testing for software in class A. Software in class B and class C requires also unit verification at the component level and software integration at the architectural level. The standard is flexible about the organization of testing by types and test stage, but coverage of requirements, risk control, usability, and test types (e.g., fault, installation, stress) should be demonstrated and documented. Finally, software release process, software maintenance process, and software risk management process documents (the last only in the case of class B and class C) are required for the certification.

This work is based on our experience regarding the development and certification (IEC 62304 standard) of a mechanical ventilator for COVID-19, called MVM (Mechanical Ventila-

tor Milano¹⁾ [17]. The development of MVM started during the first wave of the COVID-19 pandemic. In only 42 days from the initial prototype production, FDA (Food and Drug Administration) declared that MVM falls within the scope of the Emergency Use Authorization (EUA) for ventilators². EUA has been given to ventilators that the FDA determines meet specified criteria for safety, performance, and labeling. FDA recommends “that designing, evaluating, and validating these systems, is done in accordance with recognized standards for the specific device type, including the IEC 62304.”

To get an authorization for use and not just an emergency use authorization, MVM went through a software certification process. This required a re-engineering of the entire software since in the prototype the software component was underestimated, together with the production of all the required documentation. Specifically, we identified a number of activities to be undertaken to assure the quality of the software embedded in MVM, and a process of their organization able to combine the *rigidity* of the ISO standard development process with the *flexibility* of an agile attitude: the overall goal was to get the final certification in the fastest and most collaborative way but still having the rigor required by the standard.

Once produced the required documentation, the certification request has been forwarded to the Health Canada agency by the Canadian MVM manufacturer (Vexos). At the end of September 2020, MVM obtained the Health Canada Authorization³. A similar process has been carried on by the European manufacturer (Elemaster) to get the CE marking from the European Certification Authority, and this marking has been obtained at the beginning of May 2021. Thanks to these achievements, the MVM can be now sold and used in the USA, Canada, and Europe. Many countries all around the world have manifested interest in MVM ventilators. There is an ongoing project which aims to deliver MVM devices where they are most needed, and it is currently being sold by an African Union charity⁴. At the moment the ventilator is produced by Vexos, which has started the first production batch of 10,000 pieces to be delivered to the Government of Canada, and Elemaster, which has actively contributed in the project.

This work extends a previous work [18], where we reported the lessons learned matured during the software certification.

We here intend to answer to the following research question: **RQ:** *Which guidelines can be extracted from the lessons learned to help engineers during the development of critical software under emergency?* By “under emergency” we mean producing software under these two constraints: (i) *time*, meaning that the software device should be produced as soon as possible⁵; (ii) *team of volunteers*, meaning that the development team has to be rapidly established in a voluntary fashion, based

on the personal network, heterogeneous under various dimensions, and composed by people that dedicate their private time to the project, while still continuing their normal job. Note that, in other emergency situations, like hurricanes or earthquakes, there can be additional constraints like lack of energy power or Internet connection. However, we limited ourselves to the two constraints above, which are those that we observed in the experience we report.

We validated the lessons learned through feedbacks and responses to surveys received during various events and, especially, in the context of two courses for an Italian engineering society, which had 56 attendees. The feedback collected enabled us to define guidelines useful to develop critical software under emergency. They are presented in this paper.

The paper is organized as follows. Section 2 presents the research methodology we followed to perform this work. Section 3 presents the lessons learned and their validation. Section 4 presents the guidelines we derived from the lessons learned and their validation, together with the validation of the guidelines we performed. Section 5 presents the related works, and finally, the paper concludes with final remarks in Section 6.

2. Research methodology

In this section, we present the research methodology we followed. We conducted a case study [19] since it is the appropriate research methodology to study a phenomenon in its natural context, i.e., when the phenomenon is difficult to study in isolation. In our MVM project, it is difficult to clearly and precisely identify and delimit in a real context the process and the activities to be followed when producing software devices that are supposed to be compliant to safety standards under emergency.

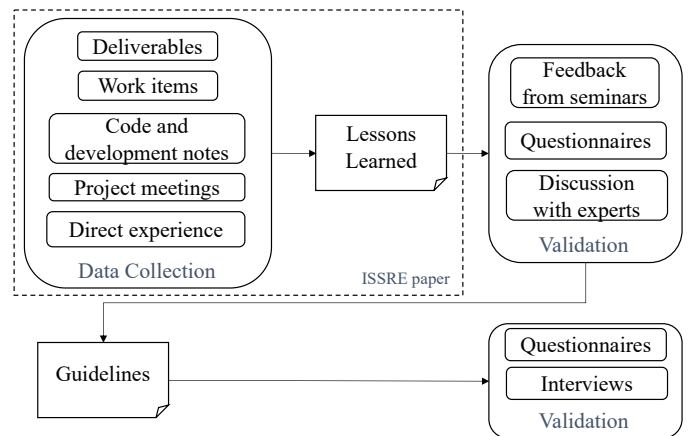


Figure 1: Research Methodology

Figure 1 provides an overview of the research methodology. As anticipated in the introduction, this paper extends a previous work [18]. As highlighted in the figure, the lessons learned have been proposed in our previous work. In this paper, we start with validation of the lessons learned through various seminars, keynotes, and lectures. This enabled us to collect precious feedback and to reason about the potential generalization

¹<http://mvm.care>

²Ventilators with EUA by FDA – search for MVM

³Health Canada Authorization

⁴<https://breathoflifeafrica.org/#MentorProject>

⁵Indeed, we might argue that time is a constraint for every company and for the production of almost every product. However, sometimes there are (emergency) situations that push even more this constraint.

of them. We also collected feedback via questionnaires that not only aimed at validating the lessons learned but we also posed additional controversial questions to better collect the opinions of the participants about the specific topic of each of the lessons learned. This enabled us to provide a set of guidelines that we then validated with experts via questionnaires and interviews.

2.1. Case description

MVM [17] is an electro-mechanical ventilator inspired by the old and reliable Manley Ventilator [20]. MVM is intended to provide ventilation support for patients that are in intensive therapy and that require mechanical ventilation. MVM works in pressure-controlled mode, i.e. the respiratory time cycle is controlled by the pressure. In particular, it implements two operative modes: Pressure Controlled Ventilation (PCV) and Pressure Support Ventilation (PSV). In the PCV mode, the respiratory cycle is kept constant and the pressure level changes between the target inspiratory pressure and the positive end-expiratory pressure. New inspiration is initiated either after a breathing cycle is over, or when the patient spontaneously initiates a breath. The PSV mode is not suitable for patients that are not able to start breathing on their own because the respiratory cycle is controlled by the patient, while MVM partially takes over the work of breathing. A new respiratory cycle is initiated with the inspiratory phase, detected by the ventilator when a sudden drop in pressure occurs. When the patient’s inspiratory flow drops below a set fraction of the peak flow, MVM stops the pressure support, thus allowing exhalation. If a new inspiratory phase is not detected within a certain amount of time (apnea lag), MVM will automatically switch to the PCV mode because it is assumed that the patient is not able to breathe alone.

A prototype of the MVM device was built in around one month and was available in the middle of April 2020. Till that moment, the team working on MVM only included physicists, physicians, and engineers as technicians and no well-defined software development process was applied, no documentation was produced, and the certification was not even pursued. However, as usually required by the governments of countries where a medical device has to be marketed, the software certification is mandatory before selling and using the device. Hence, a software task force, a group of computer scientists (including the authors), has been created to manage the software certification.

The need for certification brought to a strong re-engineering of the MVM software components, while the hardware components required only minimal changes. The whole process lasted around two months and it involved, in a continuous integration manner, all the stakeholders. The first step of the process has concerned the analysis of all the standards that must be followed during the development of a medical device, and the identification of those related to MVM. Then, a plan and a process for all the activities to be performed have been devised. Starting from the software requirement specification, we have redefined the software architecture and we have rewritten some of the MVM components. Moreover, before the deployment of the software, we have performed all the testing activities (unit, integration, and validation testing) required by the standards.

Activity	# People	Deliverables
System development plan	3	5
Supporting activities	22	12
System requirements	5	1
Software Architecture Design & Risk Management	10	3
Software Requirement Spec.	21	15
Software Detailed Design Impl.	18	N/A
Unit Testing	22	20
Integration Testing	11	2
Validation Testing	9	2

Table 1: Summary of the effort required for each phase

The MVM software is released under MIT Licence⁶, and, by following the advice of a legal office, it is stored in a private repository to control access to it and avoid incorrect usage. More information about the MVM might be found in the MVM web site⁷.

2.2. Data collection and analysis

We collected data along the various activities of the software development process, namely during requirements engineering, architectural design, testing (unit, integration, and validation), implementation, documentation, traceability checking. Table 1 summarizes the effort required in terms of (i) activities performed; (ii) number of people involved in each activity; and (iii) deliverables produced for documentation – each of these is in most cases a Microsoft Word document.

For the two months of software re-engineering and certification, activities required a large number of virtual plenary and subgroups meetings. Collected data were heterogeneous: they consist of the various deliverables created for the certification purpose and stored in a Google Drive folder – shared among the members of the certification team, authors included – but also of work items created during the project completion, such as whiteboard sketches, notes (personal or shared within subgroups), and emails. Moreover, the authors had full access to the entire source code of the project, comments in the code, changes requests, test reports, and so on.

To be used for the purpose of the case study research methodology, data coming from the sources described above, were collected independently, merged, cleaned, and stored into a folder not shared among the project members.

The collected data enabled us to formulate a set of lessons learned, which are reported in Sect. 3.

2.3. Validation of the lessons learned

The lessons learned have been discussed in three international events, where the co-authors have been invited to provide keynotes and invited presentations. Moreover, we discussed them also in lectures for PhD courses and seminars provided in three European universities. Finally, the largest validation

⁶<https://opensource.org/licenses/MIT>

⁷<http://mvm.care/>

we performed has been executed in the context of two courses for the Italian engineering society. The two courses have been performed in date September 20th 2021 and October 2nd 2021 as virtual events and had, overall, 56 attendees.

During the courses, we performed the evaluation via questionnaires (see the replication package [21]). We alternated sessions in which we were providing new content with validation sessions in which we asked the participants to answer questions. This gave us the possibility also to discuss each of the question to better grasp the various opinions and reduce our misinterpretations and biases. In the questionnaires, we checked the level of agreement and the importance of each lesson learned. We also added additional questions that have been asked before performing the evaluation of the precise lessons learned to grasp the opinions of our experts in a broader sense and without risking to influence with the lessons learned formulation. These questions were formulated in a way that they could not influence the opinion of the experts on the lessons learned. Examples are provided in Section 3. This aspect was important for us, since our evaluation of the lessons learned has been also an instrument to formulate the guidelines. The survey, composed of closed-ended questions, was spread during the entire seminar. Each of the topic of the seminar was followed by the validation of the related lessons learned. We filtered out the responses of participants who only clicked through the survey without having completed the entire survey. In total, we had 56 complete answers that we analyzed to draw conclusions.

The data collected during the validation of the lessons learned enabled us to formulate a set of guidelines that can help during the development of safety-critical systems under emergency. In Section 4 we present the guidelines, and we show how the guidelines have been synthesized from the lessons learned.

2.4. Validation of the guidelines

The guidelines have been validated via questionnaires and interviews. The questionnaire basically contains a question for each guideline asking the experts on how they agree or disagree with the specific guideline. The possible answers are in the 5 points Likert scale from 5 (Strongly Agree) till 1 (Strongly Disagree). We also added a general question at the end of the questionnaire to contain free comments from the experts. More details about the questionnaire, including the received answers, can be found in the replication package [21]. We involved experts in the development of safety-critical systems in healthcare, but also in other domains. We mainly recruited experts with which we collaborated in past projects (direct contacts) and we recruited experts in specific groups of LinkedIn, such as the “IEC 62304”, “Agile in a Regulated Environment”, “ISO 26262 Functional Safety”, or “DO-178C” groups (indirect contacts). In the case of indirect contacts, we payed particular attention to check the expertise and the soundness of the answers we got. There is no overlap between the group of people involved in the evaluation of the lessons learned and those involved in the evaluation of the guidelines. In fact, for generalization purposes, in the evaluation of the guidelines we aimed at collecting opinion also from people not specifically working in the development of safety-critical systems in healthcare.

ID	Years	Industry/Academia/ Other	Role
1	< 1	Open Source Ventilator Team	
2	1-3	Industry	Mechanical engineer
3*	4-5	Industry	CTO
4	>20	Academia	Director of Technology
5	<1	Academia	Scientist
6	11-20	Academia	Assistant professor
7	<1	Academia	PhD student
8*	<1	Academia	Assistant professor
9	4-5	Academia	Associate professor
10	<1	Academia	PostDoc
11	4-5	Academia	PostDoc
12	<1	Academia	PostDoc
13	<1	Academia	Associate professor
14	>20	Industry	CTO
15	<1	Academia	Associate professor
16*	>20	Industry	Program manager/Quality manager
17	1-3	Industry	Developer
18	6-10	Industry	System/Software architect
19	<1	Industry	Developer
20	6-10	Industry	Developer
21	11-20	Industry	System/Software architect
22	1-3	Academia	Assistant professor
23*	4-5	Academia	Associate professor
24	1-3	Academia	Assistant professor
25	6-10	Industry	Developer
26	6-10	Industry	System manager
27	11-20	Industry	Developer
28	1-3	Industry	CTO
29	11-20	Industry	CTO
30*	1-3	Industry	Developer
31	11-20	Academia	Full professor
32*	11-20	Industry	System manager
33*	>20	High risk systems	
34*	>20	High risk systems	
35	1-3	Academia	Assistant professor
36	4-5	Industry	Technical expert of software quality
37*	6-10	Industry	System/Software architect

Table 2: Experts overview. The column year stands for “Years of experience in the development of critical software”. The IDs with asterisk are experts interviewed.

Due to the heterogeneity of experts, we created identical copies of the questionnaires and distributed to various groups of experts (i.e., (i) experts in health care safety-critical systems, (ii) experts in safety-critical systems in other domains, mostly automotive, (iii) experts in agile and safety-critical systems, (iv) experts involved in the development of the MVM, (v) experts in computer science, and, finally, (vi) developers of other ventilators under emergency) to grasp different points of views. Table 2 provides an overview of the experts involved in the validation. In total we had 37 experts, which are practitioners or academics, answering the questionnaires (21 practitioners and 16 academics), and 9 of them accepted to be contacted for an interview (7 practitioners and 2 academics). Those experts that performed an interview are highlighted by a * symbol in Table 2.

As anticipated, our questionnaires mostly contain closed-ended questions, except for an optional open-ended question

Name	Description
LL.1.1	The development process was strongly influenced by the IEC 62304 standard, so the V-model, although not mandated, is the “best fit” with regulatory requirements as it produces the necessary deliverables required when seeking regulatory approval.
LL.1.2	However, it was necessary to integrate the V-model with agile practices, to combine efficiency, quality, maintainability, and flexibility.
LL.2.1	In the project, there was a quite huge overhead of coordination. The coordination of the team should not be underestimated. Open-source software development could be a good development experience from which projects of this nature can learn.
LL.2.2	Adding people is not necessarily a good solution to improve the efficiency and effectiveness of a team [22].
LL.2.3	Having responsibility for each sub-activity and setting strict intermediate goals have favored commitment and participation.
LL.3.1	The use of a great variety of tools (one tool for each particular purpose) even if not integrated and not specific for software project management, has provided indispensable support to the team.
LL.3.2	Having a partner that provided the templates and a clear review process has helped to define which activities should be performed.
LL.3.3	Standard graphical notations like UML shown to improve communication and to be easily usable by non-software experts (very skilled in other fields, though).
LL.4.1	Not having written requirements since the beginning led to having various attempts to address the requirements in different software components. Precise system requirements are very important also in an emergency situation to reduce the development time.
LL.4.2	For systems for which a prototype is present, especially if it is developed by domain experts, reverse engineering has shown to be a viable solution for discovering functionalities and configuration parameters to be included in system requirements.
LL.4.3	A traceability system helps developers to trace all the requirements and their changes through all the development process.
LL.5.1	It is important to find a balance between upfront aspects (what is planned before the start of development) and emerging aspects (decisions taken during development, e.g. by fixing wrong assumptions or making decision deliberately postponed) [23, 24].
LL.5.2	Software architecture is important even during emergency development. Without a well-defined architecture (as for the prototype), it was not clear how software components were supposed to synchronize and exchange information among them.
LL.6.1	Isolating safety-critical features, by organizing the system in different components, has allowed us to focus the safety assurance effort on a limited portion of the system.
LL.7.1	Designing a product in a modular way has been a successful decision, since, in a distributed project (such as the one of the MVM) it has allowed different teams to work in parallel on different parts of the system.
LL.7.2	We have found that using state machines, for the specification and the design, has contributed to favor the discussion on the adopted solutions even with people not used to software development since graphical representations are easily understandable.
LL.8.1	Using several programming languages in a single project is usually discouraged [25]. However, under emergency, having more languages allowed the inclusion of more developers and speed up the implementation process, with minimal effort in code integration.
LL.8.2	Sharing the coding standards and guidelines (e.g., the importance of comments [26]) with all the people involved in the implementation phase is of key importance, in particular with heterogeneous development groups, even during emergency development.
LL.8.3	State machines added flexibility and maintainability: it was very simple to make changes, regenerate, and integrate the fresh code.
LL.9.1	Isolating the critical components and defining in advance the safety classes of all the components in the developed system can significantly facilitate the testing activities, since testing can focus on the most critical parts.
LL.9.2	Besides what is required by the standards, testing activities are important when performed for safety-critical components. This is not an obvious aspect in heterogeneous teams.
LL.9.3	As MVM has been a community project, where a lot of people have worked at the same time on the same system, CI tools have proved to be crucial for maintaining under control the modifications made by all the developers.
LL.10.1	It is challenging to develop and validate systems that integrate hardware, software, and mechanics by distributed teams. Often, real hardware is needed for testing the software that is affected or affecting a piece of hardware. Software-in-the-loop simulation requires a special setting with professional simulation tools and an accurate hardware model, which is not always available and reliable.

Table 3: Lessons Learned

at the end of the two groups of questions to collect free comments from the participants. We mostly analyzed the number of given responses and analyzed the optional open-ended questions to draw conclusions. We had also the possibility to interview some of the participants of the questionnaires to better understand their answers and feedback. The interviews were driven by the questionnaires content and their answers. For each interview, we then discuss in depth the main criticalities or the stronger agreements of the specific expert. The interviews have been performed by a minimum of 4 co-authors. One co-author has the role of driver of the interview, supported by another and the other two co-authors were mainly responsible for taking notes. At the end of each interview, the co-authors met to summarize the outcomes and findings of the interview. The validation of the guidelines enabled us to discuss the benefits and risks of each of them and to clarify their scope.

2.5. Research Validity And Limitations

We discuss three types of survey validity [27].

Internal validity is concerned with the relationship between a treatment and its results and whether any unknown factors influenced the outcome of the study. To improve the instrument used in the study, we carefully designed the questionnaires and tested internally in the team of the authors.

Conclusion validity relates to the certainty that correct conclusions can be drawn about the relation between the measures and the observed outcome. To mitigate threats to conclusion validity, we included researchers and practitioners with different backgrounds in the design of the study. Moreover, we highlight that in the questionnaires for validating the lessons learned, as explained in Sec. 2.3, we had the possibility to get comments of the participants during the performed courses. For what concerns the guidelines validation, as explained in Sec. 2.4, we

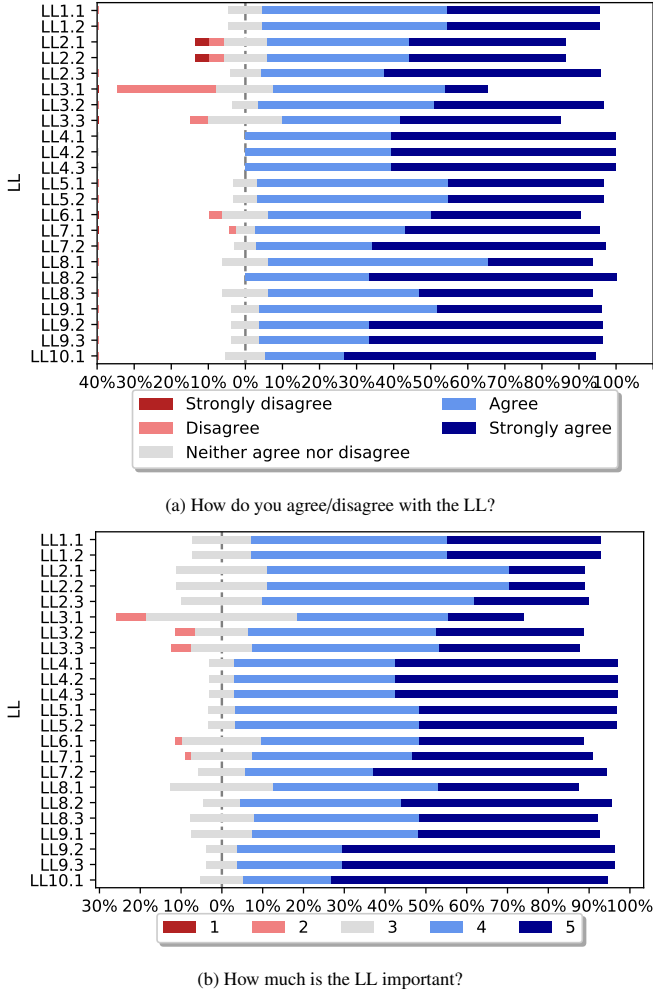


Figure 2: Validation of the lessons learned

complemented the questionnaire with interviews to understand in depth the feedback and opinion of the participants.

External validity is about the limitations of this study to generalize conclusions to other cases and domains. Our study is indeed in healthcare and in the development of devices under emergency. We made an effort to generalize the guidelines to the development of critical software in emergency. The evaluation also involved experts outside the healthcare domain, and this helped to assess the generality of the guidelines. At this point, we cannot exclude that the guidelines can be of use even when developing software not under emergency and studying how much they could be extended in a general software development is a future work.

3. Lessons Learned from the development of MVM

An in-depth description and rationale of the lessons learned can be found in [18]. In this paper, we briefly summarize them, as shown in Table 3. LL1.1-1.2 are related to the development process, while LL2.1-2.3 are related to team definition and meetings planning. LL3.1-3.3 are about supporting activities.

The remaining lessons learned relate to the development process phases: LL4.1-4.3 relate to system requirements, LL5.1-5.2 relate to software architecture design, LL6.1 relates to risk management, LL7.1-7.2 relate to software requirement analysis and software detailed design, LL8.1-8.3 relate to implementation, LL9.1-9.3 relate to unit testing, and, finally, LL10.1 relates to integration and validation testing.

After having defined the lessons learned, we have validated them with a group of experts during two seminars, by following these three steps: i) we have asked additional questions to grasp the opinions of experts before presenting lessons learned; ii) we have presented the lessons learned; iii) we have asked how much they agree with the lessons learned and how much are the lessons learned important.

The percentages of agreement/disagreement with the presented LL are reported in Fig. 2a. Generally, the participants (strongly) agreed with the LL, except in few cases. Some participants disagreed with LL2.1 (Coordination effort) and LL2.2 (Enlarging team). The former because they were skeptical of using open source software, the latter because they thought that having as many people as possible it would lead to faster results. We found also that it was confusing to ambiguously talk about teams without clearly distinguishing between coordination and development teams. LL3.1 (Multiplicity of tools) created the most disagreement, mainly because it was not clear the context where different tools would be used. Also about the use of UML (LL3.3) there were some disagreements, because other visual notation could be used, nevertheless they are not recognized as a standard in the community. For LL6.1 (Safety assurance effort) and LL7.1 (Modularity and parallelization) there were only few disagreements. For LL8.1 the validation highlights that the use of multiple programming language can lead to integration problems.

The behavior of Fig. 2a is reflected in Fig. 2b. For some participants, lessons learned with disagreement in Fig. 2a are also not that important: LL3.1 (Multiplicity of tools), LL3.3 (Use of UML), LL6.1 (Safety assurance effort) and LL7.1 (Modularity and parallelization). Exceptions are LL2.1 (Coordination effort) and LL2.2 (Enlarging team) for which the participants did not agree, although they retain them important issues. On the opposite, they agree/strongly agree with LL3.2 (use already available templates and define a clear review process), but some of the participants do not think that it is important for critical software development under emergency. As explained in the following section, Table 4 provides a description of the validation of the lessons learned; it also shows how they are mapped to the guidelines.

4. Guidelines from the development of MVM

After having gathered and validated our lessons learned through a series of dissemination activities and surveys, we have exploited the obtained results for defining a set of guidelines to be used during the development of critical systems under emergency. We have grouped, filtered, and modified the lessons learned, and, for each lesson learned, Table 4 reports

Lessons learned	Validation result	Guidelines
LL.1.1 IEC 62304 and V-Model LL.1.2 Use of agile practices	Merge the two lessons learned LL.1.1 and LL.1.2 into a unique guideline.	GL1 Plan-driven/predictive and agile integration
LL.2.1 Coordination effort LL.2.2 Enlarging team LL.2.3 Commitment and participation	Distinction between coordination team and development team in terms of responsibilities and activities.	GL4 Resources initial estimation GL5 Coordination team and plan GL6 Responsibilities assignment GL7 Flexible development teams
LL.3.1 Multiplicity of tools	Distinction between tools for coordination and communication of the coordination team and of development teams.	GL8 Inter-team coordination and communication GL9 Intra-team coordination and communication
LL.3.2 Templates and review process	Clarification on what to do when existing templates are not available and a review process is not already established.	GL2 Review process GL3 Documentation templates
LL.3.3 Use of UML	Generalization to visual/graphical notation in general	GL11 Use visual and graphical notations
LL.4.1 Written requirements LL.4.2 Reverse engineering	Merge the two LLs.	GL12 Precise requirements and reverse engineering
LL.4.3 Need of a traceability system	Recommendation of a practice (traceability system) that was not employed during the original development of the MVM but deemed to be important.	GL10 Define a traceability system
LL.5.1 Upfront aspects balancing LL.5.2 Importance of the architecture LL.7.1 Modularity and parallelization	Clarification of how to manage software architecture changes and introduction of communities of practice as an instrument to evaluate the impact on architecture and to assess and validate architectural decisions.	GL13 Define an architecture upfront GL14 Limit the upfront architecture to stable decisions GL15 Update the architecture GL16 Exploit communities of practices
LL.6.1 Safety assurance effort	Reformulation of the lesson learned.	GL17 Isolate safety-critical parts
LL.7.2 State machines for wide interpretability LL.8.3 Advantages of state machines in implementation	Identification of the two main uses of state machines.	GL18 Use state machines in specifications GL19 Use state machines for code generation
LL.8.1 Mix of programming languages	Specification of some caveats (e.g., when it does not make integration difficult).	GL20 Different programming languages
LL.8.2 Coding standards and guidelines	Reformulation of the lesson learned.	GL21 Coding standards and guidelines
LL.9.1 Testing not only safety-critical components LL.9.2 Importance of testing LL.9.3 Advantages of CI tools in community projects LL.10.1 Integration testing for SIMDs	The lessons learned were somehow overlapping and we identified the need of a better organization of them. This led to the reformulation of the lessons learned in three more clear guidelines.	GL22 Continuous integration and unit testing GL23 Focus on testing activities GL24 Role of emulators

Table 4: Mapping of lessons learned and guidelines of the development phases

how the validation activities have been used to obtain the corresponding guideline.

Overall, based on the validation of the lessons learned, in the guidelines we made a clear distinction between the coordination team, more structured and stable, and the development teams, more agile and dynamic. Moreover, we better elaborated on the architecture as a living artifact, i.e. containing both upfront aspects and aspects that emerge during development. We also exploited the validation of the lessons learned for precise and specific feedback in the guidelines' formulation.

In the following, we report the guidelines we have devised, either in general on the development process, or related to a specific development phase.

4.1. Guidelines on the development process

GL1 Plan-driven/predictive and agile integration: *Integrate plan-driven/predictive processes with agile practices, to combine rigorousness with efficiency and flexibility.* The integration of different processes allows benefiting from the good characteristics of both plan-driven [28] (or predictive [29]) and agile processes. A mix of different development processes has already been proposed. For example, the pros and cons in blending agile and waterfall processes are discussed in [30, 31], while in [32] the authors integrate agile and V-model.

GL2 Review process: *Define a clear review process to identify the activities that should be performed.* A clear review process allows to speed up the development since when a review is required before continuing with the next activities, the steps to

follow are already well known and it does not become a bottleneck. It also offers an efficient way to improve the quality and effectiveness of the development process [33].

GL3 Documentation templates: *Reuse and/or adapt existing templates, when available, for producing the documentation required by certification standards and processes, otherwise produce precise templates to be adopted by the entire project.* Considering that saving time is important, especially under emergency, it is better if there are already established templates that can be reused (for example, given by a collaborating company). This aspect is usually underestimated, especially for agile-based software development processes [34] that claim not to produce a lot of structured documentation. However, for safety-critical software, the regulations require producing a lot of deliverables that cannot be avoided. If templates are not available, it is recommended to define them before starting the development process in order to know from the beginning which information must be reported.

GL4 Resources initial estimation: *Estimate competencies, resources, and commitment of the various team members in the initial phases of the project since potential new members should be added in this initial phase.* Despite the emergency, in fact, it is important to invest some time in this upfront activity in the startup phase, as introducing members during the development process could increase the time-to-market due to the time required to understand the project. This guideline is confirmed by many other researchers, such as in [22] and [35].

GL5 Coordination team and plan: *Define the coordination team and the coordination plan of the team upfront and in the startup phase.* Despite the emergency, the coordination team needs to be ready to work as soon as possible and there must be clear indications to the development teams. Not having precise indications could lead to misunderstandings and increase the time required for completing the project. The same conclusions have been observed by [36] and [37], in which the importance of coordination teams is highlighted.

GL6 Responsibilities assignment: *Assign precise and stable responsibilities to the members of the coordination team.* We have observed that the coordination team should be as stable as possible to avoid delays that can easily propagate to the development teams. The importance of building a coordination team is well-known in the literature. In particular, most of research works consider the presence of effective leaders who both steer the development and motivate the developers crucial to ensure a successful product [38].

GL7 Flexible development teams: *Development teams can be created according to the needs during development and members should be prepared to help in various tasks according to their availability and competencies. In every iteration or sprint, team members can be assigned to different tasks.* Development teams should work agile and change should be the norm rather than the exception. Under emergency, there are string timing constraints and agility and flexibility within the development team can greatly help. The idea of flexible development teams is widely adopted in agile processes, especially when the software has to be produced for an emergency, e.g. the COVID-19 pandemic [39].

GL8 Inter-team coordination and communication: *Define the communication and coordination instruments, tools, and protocols for inter-team coordination.* When working in different groups, there is the need for clear and stable instruments, tools, and protocols for communication and coordination among the different teams. Many of the responses we have received from the surveys about LL3.2, i.e., the lesson learned from which this guideline derives, had complained about the use of multiple tools, instruments, and protocols. For this reason, here we isolate the inter-team coordination, for which the coordination mechanisms must be fixed, and the intra-team communication (see *GL9* for further details). Each project may require specific mechanisms for inter-team coordination, such as the one proposed in [40].

GL9 Intra-team coordination and communication: *Delegate to the development team members the selection of instruments and tools for development and intra-team coordination and communication.* Within development teams, members should be able to select the development, communication, and coordination instruments and tools they like. This could be considered counter-intuitive, but it can help in saving a lot of time since no training in a specific instrument or tool is necessary for the development team members. Note that the coordination and communication inter-team must be fixed a priori, as defined by *GL8*. Guaranteeing a certain grade of autonomy is a well-established principle in agile-based development teams, even if there are still a lot of challenges to be faced when implementing autonomous teams [41], mainly related to inter-team tasks. This is not always possible, e.g., when the certification standards (for example in aerospace) require the exclusive use of certified tools also for coordination.

4.2. Guidelines on the development phases

GL10 Define a traceability system: *Define upfront and in the startup phase a traceability system for the entire development.* Traceability is of key importance in the development of safety-critical systems, both for security and for certification purposes. Then, from the initial phases of the project, there is the need for a traceability system for the entire development. In fact, even under emergency, traceability information can be used to support the analysis of implications and integration of changes that occur in the system, its maintenance and evolution, and its testing activities. This guideline is recognized as valid from several works in the literature, such as [43] and [44].

GL11 Use visual and graphical notations: *Use, when possible, visual, graphical, and easy-to-understand notations, e.g. UML, for communication among team members with heterogeneous expertise and competencies.* The importance and the wide interpretability of graphical notations are universally recognized. This is even more important during an emergency, when the teams can be composed of members with different backgrounds and knowledge. In this case, graphical notations might facilitate communication among the team members [45].

GL12 Precise requirements and reverse engineering: *Write precise system requirements also in an emergency situation. If a prototype exists, use reverse engineering to extract*

Guideline	Benefits	Risks
GL1 <i>Plan-driven/predictive and agile integration</i>	<ul style="list-style-type: none"> • It enables to benefit from the good characteristics of predictive and agile processes. 	<ul style="list-style-type: none"> • It could result in an inefficient approach if advantages/disadvantages of both processes are not well known.
GL2 <i>Review process</i>	<ul style="list-style-type: none"> • It permits to clearly define the review process to speed up the development. 	<ul style="list-style-type: none"> • Misidentification of activities if inexperienced people are in charge of review process.
GL3 <i>Documentation templates</i>	<ul style="list-style-type: none"> • Precise templates permit to speed up the development while guaranteeing quality. • Reusing templates permits to save time and build on consolidated experience. 	<ul style="list-style-type: none"> • Too specialized template may not include all the information required by certification standards. • Producing precise template is expensive.
GL4 <i>Resources initial estimation</i>	<ul style="list-style-type: none"> • Despite the emergency, investing some time on this upfront activity in the startup phase permits to speed-up the development process, as well as reduce risks. 	<ul style="list-style-type: none"> • Sometimes the competencies and resources needed for a project may be unknown when it is in its initial stage, or their estimation may be not completely reliable.
GL5 <i>Coordination team and plan</i>	<ul style="list-style-type: none"> • This permits to have a coordination team ready to work and to have clear indications for the development teams. 	<ul style="list-style-type: none"> • Sometimes the project can be not well defined at the beginning, and thus it can be difficult to define upfront who to insert in the coordination team.
GL6 <i>Responsibilities assignment</i>	<ul style="list-style-type: none"> • Having the coordination team as stable as possible permits to avoid delays that can easily propagate to the development teams. 	<ul style="list-style-type: none"> • Some new tasks and responsibility may emerge during the development.
GL7 <i>Flexible development teams</i>	<ul style="list-style-type: none"> • When development teams work agile, they are ready to deal with unavoidable and frequent changes. 	<ul style="list-style-type: none"> • Moving a developer from a task to another can be difficult, if the developer is not familiar with the new one.
GL8 <i>Inter-team coordination and communication</i>	<ul style="list-style-type: none"> • Having clear and stable instruments, tools, and protocols for the communication and coordination among different teams permits to easily communicate and to focus on the development activities. 	<ul style="list-style-type: none"> • The chosen tools may be not the optimal ones for all the needs or users.
GL9 <i>Intra-team coordination and communication</i>	<ul style="list-style-type: none"> • Allowing development teams to select the development, communication, and coordination instruments for the intra-team work will permit them to work in their comfort zone. 	<ul style="list-style-type: none"> • Some certification standard requires using only certified tools during all the activities of the software life cycle. In these cases, only a limited set of certified tools should be chosen. • It can be difficult to retrieve the information at a later moment if no specific tools are used. • This freedom to choose the instruments is beneficial in the short term, but can be chaotic in the long term and in big companies.

Table 5: Benefits and risks of the guidelines on the development process

useful information. Requirements are important to guide the development and the validation phase. Even if in agile processes is common to skip or not to focus on requirement specifications, although under emergency, the requirements must be written in a precise way. This guideline is particularly useful for safety-critical systems, since certification authorities require a set of documents among which there are the requirements. The necessity of precise (even formal) requirements has been advocated for a long time, for instance in [46, 47].

GL13 Define an architecture upfront: *Define an architecture upfront to allow different teams to work in parallel on different parts of the system and to facilitate integration.* When working under emergency, every aspect that could increase the rapidness of the development is important. For this reason, a clear identification of components and interfaces might help development teams being faster, by working independently and in parallel. The architecture upfront should be stable as much as possible (see *GL14*), but teams must be ready to adapt it in case

change of requirements [48].

GL14 Limit the upfront architecture to stable decisions: *Limit the upfront architecture to stable decisions, while paying attention to concerns that matter across team borders.* An architecture description can be considered a boundary object between multiple cross-functional teams: it can be used to create a common understanding across sites while preserving each team's identity [24]. For this reason, an upfront architecture should be limited to stable decisions, and then it should be updated with emerging aspects. The upfront architecture should include aspects that influence the coordination and those interfaces that are shared among different teams (see *GL13*).

GL15 Update the architecture: *Integrate system architects into teams to capture emerging aspects during development and update the architecture accordingly.* Since the upfront architecture should be limited to stable aspects, architects, or those team members playing the role of architects and taking architectural decisions, should capture emerging aspects during development

Guideline	Benefits	Risks
GL10 <i>Define a traceability system</i>	<ul style="list-style-type: none"> It is important to properly manage traceability from the initial phases of the project. 	<ul style="list-style-type: none"> The traceability system might require tuning and adaptation during the entire development. When the traceability system is not properly defined, some people will not follow it and it will become useless.
GL11 <i>Use visual and graphical notations</i>	<ul style="list-style-type: none"> Graphical notations might facilitate the communication among team members with different background and knowledge. 	<ul style="list-style-type: none"> Focus modeling on the most important parts otherwise the return on investment will be questionable.
GL12 <i>Precise requirements and reverse engineering</i>	<ul style="list-style-type: none"> Requirements are important to guide the development and the validation phase. 	<ul style="list-style-type: none"> Writing precise requirements may require a lot of time for complex systems. If people not experienced in the field are involved, writing upfront precise requirements could be difficult.
GL13 <i>Define an architecture upfront</i>	<ul style="list-style-type: none"> A clear identification of components and interfaces might help development teams to work independently and in parallel. 	<ul style="list-style-type: none"> It can be difficult to define all the components upfront, since sometimes the need for a new component emerges during the development.
GL14 <i>Limit the upfront architecture to stable decisions</i>	<ul style="list-style-type: none"> An architecture description can be used to create a common understanding across sites while preserving each team's identity. 	<ul style="list-style-type: none"> It can become obsolete and misaligned with the implementation.
GL15 <i>Update the architecture</i>	<ul style="list-style-type: none"> This enables to enrich the upfront architecture with aspects emerging during development and update the architecture accordingly. 	<ul style="list-style-type: none"> Development team members should interact with software architects, since some emerging aspect may be not seen by them.
GL16 <i>Exploit communities of practices</i>	<ul style="list-style-type: none"> Community of practices (CoP) [42] enables architects to reason about changes and to reduce assumptions that can become inconsistencies. 	<ul style="list-style-type: none"> When the CoP is not clearly connected to the management team, it will become less effective and will only play the role of knowledge dissemination.
GL17 <i>Isolate safety-critical parts</i>	<ul style="list-style-type: none"> The isolation of safety-critical features in specific components or modules permits to limit the validation and certification activities. 	<ul style="list-style-type: none"> It could result in creating a single point of failure, which should be avoided.
GL18 <i>Use state machines in specifications</i>	<ul style="list-style-type: none"> State machines are used in various domains and are a good instrument to easily communicate complex behaviors. 	<ul style="list-style-type: none"> In general, only limited parts of a system can be modeled using state machines. Considering the artifact as a living object, it requires keeping it updated during the development.
GL19 <i>Use state machines for code generation</i>	<ul style="list-style-type: none"> The use of executable state machines promotes modifiability, maintainability, and understandability. 	<ul style="list-style-type: none"> Some certification standard may require the use of certified tools for developers, and generating code from state machines can be inapplicable in this case.
GL20 <i>Different programming languages</i>	<ul style="list-style-type: none"> When the use of different programming languages does not create integration problems, e.g. when the code is deployed on different hardware components, it would be beneficial to allow developers to use familiar languages. It allows using a language that fits better with the specific needs (e.g. to avoid using C for GUI programming). 	<ul style="list-style-type: none"> Some certification standard requires the certification of the compiler to be used by developers. In this case, using different programming languages should be avoided. If coding guidelines are followed, one should assure that they are available for all the chosen programming languages. Using various programming languages might also make more difficult code review activities.
GL21 <i>Coding standards and guidelines</i>	<ul style="list-style-type: none"> Coding standards are often required by safety standards, and in general they promote quality of the code. 	<ul style="list-style-type: none"> Following coding standards may slow up the development process if developers are not used to them.
GL22 <i>Continuous integration and unit testing</i>	<ul style="list-style-type: none"> CI tools and automatic testing instruments enable various teams to work in parallel without breaking the code, and permit to avoid the big bang integration problem. 	<ul style="list-style-type: none"> For complex systems may be difficult to initially set up the continuous integration environment.
GL23 <i>Focus on testing activities</i>	<ul style="list-style-type: none"> Safety critical components are those that require major attention. The parts of the system that are not critical can follow classic quality management recommendations. 	<ul style="list-style-type: none"> The usability of a system is affected also by non-safety-critical components, thus focusing only on the critical ones may reduce it.
GL24 <i>Role of emulators</i>	<ul style="list-style-type: none"> Simulators and emulators can speed up development and validation, but, integration, system, and acceptance testing are unavoidable. 	<ul style="list-style-type: none"> Simulators may be slightly different from the real environment (e.g., noises and interferences can be difficult to be simulated) and testing using them can be not completely reliable.

Table 6: Benefits and risks of the guidelines on the development phases.

and update the architecture accordingly [49].

GL16 Exploit communities of practices: *Exploit communities of practices to reason about changes that impact the architecture, and to assess and validate architectural decisions.* In order to reduce assumptions that can become inconsistencies, it is important to carefully assess decisions before setting them into stone. Community of practices is a good instrument for enabling architects to reason about changes [24]. They can be effectively used to solve issues that span over multiple teams [50].

GL17 Isolate safety-critical parts: *Isolate safety-critical features in specific components or modules to focus the safety assurance effort on a limited portion of the system.* Safety standards often require different levels of attention and different validation activities. The isolation of safety-critical features in specific components or modules permits limiting the validation and certification activities and, so, saving time under emergency development. The aim is similar to the well-known practice of using a security kernel with the desire to isolate and localize all “security critical” software in one place [51].

GL18 Use state machines in specifications: *Use state machines to specify modes and mode’s transitions in the requirement specification.* State machines are used in various domains and are a good instrument to easily communicate complex behaviors, especially when the development teams are heterogeneous. Moreover, if some kind of formal verification is needed, with state machines (or equivalent methods) it can be easily performed [52].

GL19 Use state machines for code generation: *Use, when possible, executable state machines for specifying the main functional logic and the critical part of the system, and, then, generate code from the state machines.* Based on our experience, the use of executable state machines promotes modifiability, maintainability, and understandability (see *GL18*). In particular, tools like Yakindu SCT⁸ or other state-machine-based tools can be a good choice when it comes to developing a critical part of the system, as they allow generating automatically actual code [53, 54], which can be verified and tested at state-machine-side [55].

GL20 Different programming languages: *Allow the use of different programming languages to facilitate the inclusion of heterogeneous developers and speed up the implementation process, when this does not create integration problems.* When the use of different programming languages does not create integration problems, e.g. when code produced with different languages is deployed on different hardware components or when the communication mechanisms between modules are language-independent, it would be beneficial to allow development teams to use languages they are familiar with [25]. As explained in LL8.1 in [18], when developing software under emergency, with heterogeneous development teams, exploiting the competencies that every member has on a particular programming language can aid in reducing the time required for the completion of the project.

GL21 Coding standards and guidelines: *Adopt coding standards and guidelines from the beginning.* Safety standards

often require coding standards and in general, they promote the quality of the code. Using coding standards and guidelines can increase the readability of the code, which is important for code inspections and static analysis. Moreover, even under emergency, if the composition of the development teams is heterogeneous defining in advance coding standards and guidelines is useful for preventing possible errors or hardly-understandable code. The importance of adhering to coding standards from the beginning is highlighted by many researchers, such as in [56], where the authors empirically assess the value of coding standards and suggest not to insert them at a later time as any modification (aimed at adapting the software to the chosen coding standards) has a non-zero probability of introducing a fault or triggering a previously concealed one.

GL22 Continuous integration and unit testing: *Use CI tools and automated unit testing in order to continuously integrate the contributions of the various teams, to keep and promote quality, and to maintain under control the modifications made by all the developers.* CI tools and automatic testing instruments enable various teams to work in parallel without breaking the code, and permit to avoid the big bang integration problem. The importance of continuous integration is highlighted in [57], where CI tools are claimed to be effective since they allow a shorter time between the possible introduction of a bug in the system and its detection. This is of paramount importance, especially for complex safety-critical systems developed under emergency and in a distributed way.

GL23 Focus on testing activities: *While guaranteeing the quality of the whole system, focus the testing activities to safety-critical components as required by the standard.* Safety critical components are those that require major attention, in terms of quality and test effort. The parts of the system that are not critical can follow classic quality management recommendations. Focusing software testing activities on critical components is often used in practice, especially when companies want to reduce their time-to-market [58]. Anyhow, testing is important on all the components of a safety-critical device, regardless of the safety classification.

GL24 Role of emulators: *When possible, use simulators and/or emulators, but plan for an integration, system, and acceptance testing phase.* Simulators and emulators can speed up the development and validation, but they are often limited and integration, system, and acceptance testing cannot be performed using them. This is a well known aspect of integration testing for safety-critical systems, such as for train and ships [59], or automotive [60]. Especially if the system under development is composed of hardware and software, the *testing in the field* activity must be planned and performed in the real environment, with the real hardware.

4.3. Guidelines validation

After having defined the guidelines, we have validated them in two different ways, i.e., through questionnaires and interviews as explained in Sect. 2.4. During the questionnaires, the participants have been asked about their agreement with each guideline, using a Likert scale (from strongly disagree, to strongly agree), the results of these surveys are presented in

⁸<https://www.itemis.com/en/yakindu/state-machine/>

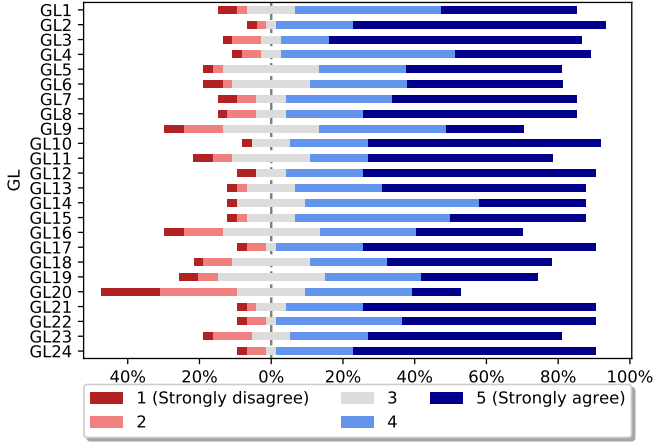


Figure 3: How do you agree/disagree with the GL?

Fig. 3. The agreement levels over the guidelines are generally lower than those over the lessons learned (see Fig. 2a). This was somehow expected since the guidelines are given in more affirmative and prescriptive style. Moreover, this is also due to the wider experience and expertise of the involved experts. In fact, we decided to involve international practitioners with various expertise, as explained in Section 2, with the aim of collecting different points of view. In all the cases (except for GL20), the agreement levels remain very positive.

To further investigate the results obtained during the questionnaires, we have conducted several interviews (see Sect. 2.4) to examine the reasons for which the participants were particularly agreeing or disagreeing with our guidelines. Tab. 5 and Tab. 6 summarize the opinions we have collected in terms of benefits expected if one follows the guideline and what are the risks. Especially for those with lower agreement (like GL20), we were able, thanks to the direct interaction with the experts, to better identify the limits and circumscribe their intended use. This will enable users of the guidelines to carefully assess how each guideline should be applied to their project. For instance, for GL20 we have identified a typical scenario in which it can be followed, when different programming languages are used on different hardware components.

5. Related works

In this section, we (i) analyze papers that describe the role of software systems in helping manage the COVID-19 pandemic, and (ii) compare our guidelines with works exploring the use of agile practices for the development of safety-critical systems.

5.1. Software systems and COVID-19

In the literature, we might find studies and experience reports on the use of existing software and IT approaches for managing the COVID-19 emergency; there are also cases of application of software systems to COVID-19 patients [61, 62]. However, no particular attention is given to software development. Other studies are focused on developing new medical software but in

a classical software development environment [63, 64]. Some suggestions like the integration of agile practices and the need for rigorous requirement specification have been applied to our project as well, but we expect that medical software development without any emergency is more planned and structured, and some of the lessons we learned may not hold.

Other studies describe the development of new software and AI solutions for COVID-19 during the emergency. The area of medical imaging is, as expected, the most relevant [65]. Machine Learning algorithms have been promptly adapted to the diagnosis of COVID-19 cases. In [66], the authors present a rapid AI development cycle for an automated detection solution using deep learning CT Image Analysis. Those papers focus more on data collection and management than software development. Regarding, instead, the SW development of medical devices for COVID-19, we can compare MVM with the numerous open-source community-driven projects working on mechanical ventilators [67]. We could not find reports on their experience in SW development and certification. Not surprisingly, only a few of them⁹ reached the certification while many others, even if very promising, are still behind. We believe that this paper could provide a guideline for them as it would help, God forbid, future similar projects.

5.2. Safety-critical systems and agile development practices

Agile development practices might be seen in antithesis with the development of safety-critical systems. However, there are various attempts to bring benefits of agility and flexibility of agile practices to more rigorous, stage-gate, and predictive development methodologies.

In automotive, rigorous, stage-gate, and predictive processes have previously been the norm, with requirements elicited and described at the beginning of the project and with architecture created mainly during an early phase and then used to guide subsequent development phases [68, 69, 23]. However, vehicles are becoming software-intensive complex systems, they are shocked by various emerging business and technological needs like electrification, autonomy, connectivity, and they are increasingly expected to evolve continuously. This drives the automotive domain towards more exploratory, iterative, and agile ways of working [70, 71, 68, 69]. To testify the importance of the topic in the automotive domain, we mention EchoScrum¹⁰, which is an agile development model specifically designed to ensure compliance with ISO 26262, ISO 21434 (Cyber Security), and ISO/PAS 21448 (SOTIF).

Due to safety and legal concerns, requirements must be properly elicited, analyzed, and described [68]. This is aligned with our guideline *GLI2*. However, traditional ways of working are no longer sufficient for various reasons [68]. Similarly to what we do recommend in guideline *GLI7*, the work in [68] recommends to focus requirement efforts where crucial, e.g., on safety-critical functionality. Aligned with guidelines *GLI1*, *GLI8*, and *GLI9*, the authors recommend to use model-based

⁹<https://bit.ly/2P0Pm3g>

¹⁰<https://fsq-experts.com/products/>

RE and especially executable models for having early feedback. Aligned with guidelines *GL13* and *GL14*, the authors of [68] recommend postponing and delegate some decisions to developers. This aspect will be further explored in the remainder of this section, when we will discuss the role of architecture in agile. Aligned with guideline *GL22*, in paper [68], the authors recognize that test automation is essential for CI. They also touch traceability that we address in guideline *GL10*. However, the authors highlight that current solutions are not satisfactory for the industrial needs. The work in [72] highlights that existing traceability management approaches show their limitations when integrated in constantly changing development contexts involving multiple stakeholders: there is the need of flexible tool solutions, support for varying levels of data quality, change propagation and versioning facilities, as well as traceability covering also the organization part. In our work, we did not go so deep in the investigation of traceability aspects, but it would be interesting in future work to better analyze these aspects.

Traditionally, software architectures are mostly produced upfront and provide a blueprint for the entire development. More recently, there is a transition toward “just-in-time architecture” [23]: it is recognized that some upfront is needed and that “a clear and well-defined architecture facilitates and enables agility” [23], since it facilitates integration while providing flexibility to the various agile teams working in parallel. This is much in line with our guideline *GL13*. However, it is also recognized that architecture is not a phase of development [73], but, instead, some decisions and aspects might emerge during development and it is important to capture these emerging elements to keep the architecture description consistent with the implementation [24]. This is also confirmed by our guidelines *GL14* and *GL15*.

As anticipated, when transitioning towards agility in the development of software-intensive complex systems, there is the need for scaling agility beyond teams, and this requires specific strategies to support such scaling, and it has also implications on the organizational structure [74]. This has been found challenging [75, 76]; one of the main impediment to provide engineers with continuous feedback on system level is the lack of infrastructure and tools [77]. Useful practices for scaling agile, e.g. set-based design [78] and avoiding narrow product definitions [79], have been promoted by scaled agile frameworks, e.g. SAgile¹¹ [78] and LeSS¹². Another approach to scale benefits of agile approaches beyond teams are boundary objects [80, 49], i.e. objects that are relevant for more than one team and then allow for aligning across organizational boundaries.

6. Final remarks

In this paper, we report our experience in the certification of the software of a mechanical ventilator that has been produced by a team of researchers all around the world during the

first wave of the COVID-19 pandemic. Out of this experience, we synthesized a set of lessons learned that might help other researchers to develop and certify software under emergency, i.e. with strict time constraints and with volunteers dedicating their private time to a social project. We validated the lessons learned with 56 experts during a professional course we held. We made use of questionnaires, where each question was followed by a consequent discussion. The validation of the lessons learned enabled us to extract from them a set of 24 guidelines spanning from the development process, development phases, and people management and coordination. Then, through questionnaires and interviews, we validated the guidelines with academic and industrial experts with different profiles: (i) experts in developing software for health care devices, (ii) experts in developing safety-critical software for automotive, and (iii) experts in combining agile methods with rigorous methodologies for the development of safety-critical software. The variety of typologies of experts enabled us to collect different points of view and opinions about the guidelines. Our guidelines offer concrete support for engineers during the development of critical software under emergency. Each guideline is offered with a discussion of its benefits and risks. In this way, engineers can decide and evaluate whether a specific guideline is appropriate in their specific context. Our study opens to the possibility of using agile methods together with document-driven and stricter development processes. Part of the guidelines stem from agile practices (e.g., *GL4* to *GL9*, and *GL14*), while others come from stricter processes (e.g., *GL12*, *13*, *18*, *21*). Others already strive to combine the two (e.g. *GL1*). Studying this topic is of great interest and brings up several questions: How much the two approaches can be combined and what is the best way? Are there conflicts (for example *GL13* vs *GL15*)? Can we define specific guidelines on how to combine elements of both “worlds”? Precise answers to these questions will be provided in our future works.

Acknowledgement

This work is partially supported by the FIRS Italian project MVM-Adapt (Milano Ventilatore Meccanico Adattativo in presenza d’incertezza), FIRS2020IP_05310. The authors would like to thank the entire team who contributed to the realization of the MVM project. We would like to thank also the experts that helped us in the validation of lessons learned and guidelines.

References

- [1] A. Kornecki, J. Zalewski, Software certification for safety-critical systems: A status report, in: 2008 International Multiconference on Computer Science and Information Technology, 2008.
- [2] G. Ferreira, C. Kästner, J. Sunshine, S. Apel, W. L. Scherlis, Design dimensions for software certification: A grounded analysis, CoRR abs/1905.09760 (2019).
- [3] A. Gannous, A. Andrews, Integrating safety certification into model-based testing of safety-critical systems, in: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), 2019.
- [4] A. Wölfl, N. Siegmund, S. Apel, H. Kosch, J. Krautlager, G. Weber-Urbina, Generating qualifiable avionics software: An experience report (e), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015.

¹¹<https://www.scaledagileframework.com/>

¹²<https://less.works/less/framework>

- [5] A. Wölfl, Data management in certified avionics systems, Ph.D. thesis, Universität Passau (2018).
- [6] A. Hovsepian, D. Van Landuyt, S. Op de beeck, S. Michiels, W. Joosen, G. Rangel, J. Fernandez Briones, J. Depauw, Model-driven software development of safety-critical avionics systems: an experience report, Vol. 1249, 2014.
- [7] R. Pietrantuono, S. Russo, Robotics software engineering and certification: Issues and challenges, in: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2018.
- [8] J. L. de la Vara, E. Parra, L. Alonso, R. Mendieta, B. López, J. M. Álvarez-Rodríguez, Integration of tool support for assurance and certification and for knowledge-centric systems engineering, in: IEEE Int. Symp. on Software Reliability Eng. Workshops (ISSREW), 2019.
- [9] N. Hrgarek, Certification and regulatory challenges in medical device software development, in: 2012 4th International Workshop on Software Engineering in Health Care (SEHC), 2012, pp. 40–43.
- [10] S. Pelayo, S. Bras Da Costa, N. Leroy, S. Loiseau, M.-C. Beuscart-Zéphir, Software as a medical device: Regulatory critical issues, *Studies in health technology and informatics* 183 (2013) 337–42.
- [11] I. Lee, G. J. Pappas, R. Cleaveland, J. Hatcliff, B. H. Krogh, P. Lee, H. Rubin, L. Sha, High-confidence medical device software and systems, *Computer* 39 (4) (2006) 33–38.
- [12] M. N. K. Boulos, A. C. Brewer, C. Karimkhani, D. B. Buller, R. P. Dellavalle, Mobile medical and health apps: state of the art, concerns, regulatory control and certification, *Online Journal of Public Health Informatics* 5 (3) (Feb. 2014).
- [13] J. Neto, J. Damásio, P. Monthaler, M. Morais, Product-based safety certification for medical devices embedded software, *Studies in health technology and informatics* 216 (2015).
- [14] W. J. Gordon, A. D. Stern, Challenges and opportunities in software-driven medical devices, *Nature Biomedical Engineering* 3 (7) (2019).
- [15] A. Ohne Autor Fd, General Principles of Software Validation; Final Guidance for Industry and FDA Staff, V.2.0, FDA document formal (2002).
- [16] IEC 62304 Medical device software — Software life cycle processes.
- [17] A. Abba, et al., The novel mechanical ventilator milano for the COVID-19 pandemic, *Physics of Fluids* 33 (3) (2021) 037122.
- [18] A. Bombarda, S. Bonfanti, C. Galbiati, A. Gargantini, P. Pelliccione, E. Riccobene, M. Wada, Lessons learned from the development of a mechanical ventilator for COVID-19, in: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), 2021.
- [19] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, *Empirical software engineering* 14 (2) (2009) 131.
- [20] R. N. Westhorpe, C. Ball, The manley ventilator, *Anaesthesia and intensive care* 40 (5) (2012) 749–750.
- [21] A. Bombarda, S. Bonfanti, C. Galbiati, A. Gargantini, P. Pelliccione, E. Riccobene, Replication package. available online at: <https://se4med.github.io/GuidelinesForCriticalSoftware/> (2022).
- [22] F. P. Brooks, *The Mythical Man Month*, Prentice Hall, 1995.
- [23] P. Pelliccione, E. Knauss, R. Heldal, M. Magnus Ågren, P. Mallozzi, A. Alming, D. Borgentun, Automotive architecture framework: The experience of volvo cars, *Journal of Systems Architecture* 77 (2017).
- [24] R. Wohlrab, U. Eliasson, P. Pelliccione, R. Heldal, Improving the consistency and usefulness of architecture descriptions: Guidelines for architects, in: 2019 IEEE Int. Conf. on Software Architecture (ICSA), 2019.
- [25] P. Mayer, M. Kirsch, M. A. Le, On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers, *Journal of Software Engineering Research and Development* 5 (1) (2017).
- [26] J. Raskin, Comments are more important than code: The thorough use of internal documentation is one of the most-overlooked ways of improving software quality and speeding implementation., *Queue* 3 (2) (2005).
- [27] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, A. Wessln, *Experimentation in Software Engineering*, 2012.
- [28] B. Boehm, R. Turner, Balancing agility and discipline: evaluating and integrating agile and plan-driven methods, in: *Proceedings. 26th International Conference on Software Engineering*, 2004.
- [29] B. Meyer, *Agile!*, Springer International Publishing, 2014.
- [30] C. Quist, Benefits of blending agile and waterfall project planning methodologies, Tech. rep., University of Oregon (2015).
- [31] R. J. Kusters, Y. van de Leur, W. G. M. M. Rutten, J. J. M. Trienekens, When agile meets waterfall, in: 19th Int. Conf. on Enterprise Information Systems, 2017.
- [32] M. McHugh, F. McCaffery, G. Coady, An agile implementation within a medical device software organisation, in: *Software Process Improvement and Capability Determination*, 2014.
- [33] P. Kess, H. Haapasalo, Knowledge creation through a project review process in software production, *International Journal of Production Economics* 80 (1) (2002) 49–55, innovation of Technology Management. doi:[https://doi.org/10.1016/S0925-5273\(02\)00242-6](https://doi.org/10.1016/S0925-5273(02)00242-6). URL <https://www.sciencedirect.com/science/article/pii/S0925527302002426>
- [34] J. A. Livermore, Factors that significantly impact the implementation of an agile software development methodology., *J. Softw.* 3 (4) (2008).
- [35] T. Marks, Accelerating the project, in: *The Practitioner Handbook of Project Controls*, 2020.
- [36] N. B. Moe, T. Dingsøyr, K. Rolland, To schedule or not to schedule? an investigation of meetings as an inter-team coordination mechanism in large-scale agile software development (2018).
- [37] A. Subbarao, M. Mahrin, A systematic review of coordination approaches and indicators in global software development projects, *Journal of Advanced Research in Dynamical and Control Systems* 11 (10) (2019).
- [38] Y. Li, C.-H. Tan, H.-H. Teo, Leadership characteristics and developers’ motivation in open source software development, *Information and Management* 49 (5) (2012) 257–267. doi:10.1016/j.im.2012.05.005.
- [39] J. a. Varajão, Software development in disruptive times: Creating a software solution with fast decision capability, agile project management, and extreme low-code technology, *Queue* 19 (1) (2021).
- [40] H. Nyrud, V. Stray, Inter-team coordination mechanisms in large-scale agile, in: *Proceedings of the XP2017 Scientific Workshops*, 2017.
- [41] V. Stray, N. B. Moe, R. Hoda, Autonomous agile teams: Challenges and future directions for research, in: *Proceedings of the 19th International Conference on Agile Software Development: Companion*, 2018.
- [42] Scaled Agile, Communities of practice, available online at <https://www.scaledagileframework.com/communities-of-practice/> (2022).
- [43] G. Spanoudakis, A. Zisman, Software Traceability: A Roadmap, in: *Handbook Of Software Engineering And Knowledge Engineering*, 2005.
- [44] C. Ingram, S. Riddle, Cost-benefits of traceability, in: *Software and Systems Traceability*, 2011.
- [45] M. R. V. Chaudron, W. Heijstek, A. Nugroho, How effective is UML modeling ?, *Software & Systems Modeling* 11 (4) (2012).
- [46] D. L. Parnas, J. Madey, Functional documents for computer systems, *Science of Computer Programming* 25 (1) (1995) 41–61.
- [47] P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoo, E. Riccobene, Integrating formal methods into medical software development: The ASM approach, *Science of Computer Programming* 158 (2018) 148–167. doi:10.1016/j.scico.2017.07.003.
- [48] J. Eckstein, Architecture in large scale agile development, in: *Lecture Notes in Business Information Processing*, 2014.
- [49] R. Wohlrab, P. Pelliccione, E. Knauss, M. Larsson, Boundary objects and their use in agile systems engineering, *Journal of Software: Evolution and Process* 31 (5) (2019).
- [50] T. Kahkonen, Agile methods for large organizations - building communities of practice, in: *Agile Development Conference*, 2004, pp. 2–10.
- [51] J. M. Rushby, Design and verification of secure systems, *ACM SIGOPS Operating Systems Review* 15 (5) (1981).
- [52] P. Arcaini, A. Bombarda, S. Bonfanti, A. Gargantini, E. Riccobene, P. Scandurra, The ASMETA Approach to Safety Assurance of Software Systems, 2021.
- [53] A. Bombarda, S. Bonfanti, A. Gargantini, Developing medical devices from abstract state machines to embedded systems: A smart pill box case study, in: *Software Technology: Methods and Tools*, Springer International Publishing, 2019, pp. 89–103. doi:10.1007/978-3-030-29852-4_7.
- [54] A. Bombarda, S. Bonfanti, A. Gargantini, E. Riccobene, Developing a prototype of a mechanical ventilator controller from requirements to code with ASMETA, *Electronic Proceedings in Theoretical Computer Science* 349 (2021) 13–29. doi:10.4204/eptcs.349.2.
- [55] F. Wagner, *Modeling Software with Finite State Machines*, Auerbach Publications, 2006.
- [56] C. Boogerd, L. Moonen, Assessing the value of coding standards: An em-

- pirical study, in: 2008 IEEE Int. Conf. on Software Maintenance, 2008.
- [57] P. Duvall, S. Matyas, A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, 2007.
 - [58] M. Felderer, R. Ramler, Risk orientation in software testing processes of small and medium enterprises: an exploratory and comparative study, *Software Quality Journal* 24 (3) (Aug. 2015).
 - [59] C. Dufour, G. Dumur, J.-N. Paquin, J. Belanger, A pc-based hardware-in-the-loop simulator for the integration testing of modern train and ship propulsion systems, in: 2008 IEEE Power Electr. Specialists Conf., 2008.
 - [60] J. Schroeder, C. Berger, T. Herpel, Challenges from integration testing using interconnected hardware-in-the-loop test rigs at an automotive oem: An industrial experience report, in: Proceedings of the First International Workshop on Automotive Software Architecture, WASA '15, Association for Computing Machinery, 2015.
 - [61] B. A. Jr, Use of telemedicine and virtual care for remote treatment in response to covid-19 pandemic, *J. Medical Syst* 44 (7) (2020) 132.
 - [62] A. Asadzadeh, S. Pakkhoo, M. M. Saeidabad, H. Khezri, R. Ferdousi, Information technology in emergency management of covid-19 outbreak, *Informatics in Medicine Unlocked* 21 (2020) 100475.
 - [63] C. Denger, R. L. Feldmann, M. Host, C. Lindholm, F. Shull, A snapshot of the state of practice in software development for medical devices, in: First Int. Symp. on Emp. Software Eng. and Measurement (ESEM), 2007.
 - [64] M. McHugh, O. Cawley, F. McCaffry, I. Richardson, X. Wang, An agile V-model for medical device software development to overcome the challenges with plan-driven software development lifecycles, in: 2013 5th Int. Work. on Software Engineering in Health Care (SEHC), 2013.
 - [65] F. Shi, J. Wang, J. Shi, Z. Wu, Q. Wang, Z. Tang, K. He, Y. Shi, D. Shen, Review of artificial intelligence techniques in imaging data acquisition, segmentation, and diagnosis for COVID-19, *IEEE Reviews in Biomedical Engineering* 14 (2021).
 - [66] O. Gozes, M. Frid-Adar, H. Greenspan, P. D. Browning, H. Zhang, W. Ji, A. Bernheim, E. Siegel, Rapid ai development cycle for the coronavirus (covid-19) pandemic: Initial results for automated detection & patient monitoring using deep learning ct image analysis (2020). *arXiv:2003.05037*.
 - [67] J. M. Pearce, A review of open source ventilators for COVID-19 and future pandemics, *F1000Research* 9 (2020) 218.
 - [68] S. M. Ågren, E. Knauss, R. Heldal, P. Pelliccione, G. Malmqvist, J. Bodén, The impact of requirements on systems development speed: a multiple-case study in automotive, *Req. Engineering* 24 (3) (2019).
 - [69] S. M. Ågren, E. Knauss, R. Heldal, P. Pelliccione, A. Alming, M. Antonsson, T. Karlkvist, A. Lindeborg, Architecture evaluation in continuous development, *Journal of Systems and Software* 184 (2022).
 - [70] P. Hohl, J. Münch, K. Schneider, M. Stupperich, Real-life challenges on agile software product lines in automotive, in: Proc. of Int. Conf. on Product-Focused Software Process Improvement (PROFES), 2017.
 - [71] M. Stupperich, S. Schneider, Process-focused lessons learned from a multi-site development project at daimler trucks, in: Proc. of 6th Int. Conf. on Global Software Engineering (ICGSE), 2011.
 - [72] R. Wohlrab, P. Pelliccione, A. Shahrokni, E. Knauss, Why and how your traceability should evolve: Insights from an automotive supplier, *IEEE Software* 38 (4) (2021).
 - [73] R. N. Taylor, N. Medvidovic, E. M. Dashofy, *Software Architecture: Foundations, Theory and Practice*, Addison-Wesley, 2007.
 - [74] S. M. Ågren, R. Heldal, E. Knauss, P. Pelliccione, Agile beyond teams and feedback beyond software in automotive systems, *IEEE Transactions on Engineering Management* (2022).
 - [75] R. Kasauli, E. Knauss, J. Horkoff, G. Liebel, F. G. de Oliveira Neto, Requirements engineering challenges and practices in large-scale agile system development, *Systems and Software* 172 (2021).
 - [76] R. Kasauli, E. Knauss, J. Nakatumba-Nabende, B. Kanagwa, Agile islands in a waterfall environment: Requirements engineering challenges and strategies in automotive, in: Proc. of Int. Conf. on Evaluation and Assessment in Software Engineering (EASE), 2020.
 - [77] E. Knauss, P. Pelliccione, R. Heldal, M. Ågren, S. Hellman, D. Maniette, Continuous integration beyond the team: a tooling perspective on challenges in the automotive industry, in: 10th ACM/IEEE Int. Symp. on Empirical Software Eng. and Measurement, 2016.
 - [78] R. Knaster, D. Leffingwell, *SAFe 4.0 Distilled: Applying the Scaled Agile Framework for Lean Software and Systems Engineering*, Addison-Wesley Professional, 2017.
 - [79] C. Larman, B. Vodde, *Large-scale scrum: More with LeSS*, Addison-Wesley Professional, 2016.
 - [80] T. Sedano, P. Ralph, C. Péraire, The product backlog, in: IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019.