# Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study

FIORELLA ZAMPETTI, University of Sannio
DAMIAN TAMBURRI, Eindhoven University of Technology
SEBASTIANO PANICHELLA, Zurich University of Applied Sciences
ANNIBALE PANICHELLA, Technical University of Delft
GERARDO CANFORA and MASSIMILIANO DI PENTA, University of Sannio

Continuous Integration and Delivery (CI/CD) practices have shown several benefits for software development and operations, such as faster release cycles and early discovery of defects. For Cyber-Physical System (CPS) development, CI/CD can help achieving required goals, such as high dependability, yet it may be challenging to apply. This article empirically investigates challenges, barriers, and their mitigation occurring when applying CI/CD practices to develop CPSs in 10 organizations working in eight different domains. The study has been conducted through semi-structured interviews, by applying an open card sorting procedure together with a member-checking survey within the same organizations, and by validating the results through a further survey involving 55 professional developers. The study reveals several peculiarities in the application of CI/CD to CPSs. These include the need for (i) combining continuous and periodic builds while balancing the use of Hardware-in-the-Loop and simulators, (ii) coping with difficulties in software deployment (iii) accounting for simulators and Hardware-in-the-Loop differing in their behavior, and (vi) combining hardware/software expertise in the development team. Our findings open the road toward recommenders aimed at supporting the setting and evolution of CI/CD pipelines, as well as university curricula requiring interdisciplinarity, such as knowledge about hardware, software, and their interplay.

CCS Concepts: • **Software and its engineering → Maintaining software;**

Additional Key Words and Phrases: Continuous Integration and Delivery, Cyber-Physical Systems, empirical software engineering

This work was supported by Horizon 2020 (EU Commission) for the project *COSMOS* (DevOps for Complex Cyber-physical Systems), project number 957254-COSMOS.
Authors' addresses: F. Zampetti, G. Canfora, and M. Di Penta, University of Sannio, Via Traiano, 9, Benevento, Italy, 82100; emails: fiorellazampetti@gmail.com, {canfora, dipenta}@unisannio.it; D. Tamburri, Eindhoven University of Technology and the Jheronimus Academy of Data Science (JADS), St. Janssingel 92, ‘s-Hertogenbosch, The Netherlands, 5211 DA; email: d.a.tamburri@tue.nl; S. Panichella, Zurich University of Applied Sciences, Steinberggasse 13, Winterthur, Switzerland, 8400; email: panc@zhaw.ch; A. Panichella, Technical University of Delft, Van Mourik Broekmanweg 6, Delft, The Netherlands, 2628 XE; email: a.panichella@tudelft.nl.

1049-331X/2023/04-ART73 $15.00
https://doi.org/10.1145/3571854

**ACM Reference format:**
Fiorella Zampetti, Damian Tamburri, Sebastiano Panichella, Annibale Panichella, Gerardo Canfora, and Massimiliano Di Penta. 2023. Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 73 (April 2023), 44 pages.
https://doi.org/10.1145/3571854

## 1 INTRODUCTION

**Cyber-Physical Systems (CPSs)** comprise heterogeneous software and hardware components interacting with each other. They aim at automating operations in different domains, such as automotive, aerospace, healthcare, or railways. As it happens for any software system, CPSs continuously evolve to cope with new customer requirements and technology changes. However, CPSs require a tailored development and operation (DevOps) process and are more challenging to evolve than conventional software [32, 51, 73, 74].

In such a context, adopting effective **Continuous Integration and Delivery (CI/CD)** practices off the DevOps menu is extremely relevant for setting the execution environment, such as **Hardware-in-the-Loop (HiL)** or simulators. Even though CI/CD has been found to be effective in introducing several advantages in software development, such as the reduction of release cycles and the early discovery of defects [75], its application implies overcoming barriers and challenges [9, 33].

When enacting CI/CD for CPSs, it is expected that further barriers and challenges will arise. In general, existing CI/CD technology cannot be applied to CPSs as is [39]. On the one hand, CPSs demand suitable **Verification & Validation (V&V)** techniques, and the interaction with HiL or the need to replace them with suitable mock-ups or simulators make the application of CI/CD challenging at best. On the other hand, although for conventional software systems good and bad practices for applying CI/CD have been defined [15, 85], for what concerns CPSs, the practice is still immature to be able to do so. Specifically, the combination of (very diversified and evolving) hardware devices and software, the complex execution scenarios, and the need for simulating hardware components during some build stages introduce new facets that must be considered when setting up a CPS development process, and particularly CI/CD pipelines for CPS development.

This article aims to empirically investigate the challenges and barriers practitioners encounter while setting up and maintaining a CI/CD pipeline for CPSs, as well as the mitigation strategies adopted to deal with them. Specifically, the study has been conducted through (i) semi-structured interviews with 10 industrial practitioners involved in CPS development for eight different domains (i.e., aerospace, automotive, energy, healthcare, railways, robotics, identification technology (i.e., **Radio Frequency IDentification (RFID)**, and acoustic sensors), (ii) by applying open coding [35] and card sorting [68] to the interview transcripts, (iii) by conducting a member-checking survey within the same organizations involved in the interviews aiming at corroborating the relations between challenges/barriers and mitigation strategies, and (iv) by assessing the relevance of the identified challenges/barriers and related mitigation through a survey involving 55 practitioners involved in CPS development for nine different domains.

We start by characterizing the CI/CD practices of the interviewed organizations, focusing more on their build automation processes. In doing this, we target three aspects of CI/CD for CPSs, namely (i) the pipeline setting, (ii) the involved phases (static analysis, testing, delivery, etc.), and (iii) the usage and configuration of simulators and/or HiL. After that, we look at challenges and barriers the organizations encounter, as well as mitigation strategies being adopted to deal with them.

The elicited set of challenges, barriers, and mitigation strategies are impactful by providing insights to project leaders and developers, guiding them to configure CI/CD pipelines for CPSs, as well as to staff projects properly to coordinate resources with different skills and expertise and acquire equipment. Furthermore, results highlight directions in which education for CPS development must be improved. This includes not only covering interdisciplinary topics between software development, measurements, and automated control but also a proper introduction to software architectures and design principles, making CPS development flexible enough when switching between simulators and HiL. Last but not least, we identify areas where further research is required, among others domain-aware decision making, the integration of simulators and HiL in the pipeline, and further research in the area of test automation and flakiness detection/avoidance. The specificity of each CPS not only makes some lessons hard to generalize (each pipeline tends to be very different from others) but also poses challenges when leveraging **Machine Learning (ML)** approaches upon developing recommender systems.

*Article Structure.* The rest of the article is organized as follows. As a basis for the study, Section 2 discusses the relevant literature. Section 3 describes the study methodology, whereas Section 4 reports and discusses the study results. Section 5 details the study implications, whereas threats to the study validity are discussed in Section 6. Finally, Section 7 concludes the article and outlines future directions.

The study material (after redacting interview transcripts) is available online [84].

## 2 RELATED WORK

This section discusses the literature related to (i) CPS development leveraged for the inception of our study, (ii) CI/CD process, and (iii) CI/CD good and bad practices. Note that this is not an exhaustive systematic literature review on the topic, but rather it points out papers discussing challenges in CPS development and in CI/CD. Finally, it is important to highlight that although challenges related to CPS development are already investigated from previous literature, to the best of our knowledge there is very limited empirical evidence on how such challenges translate when setting a CI/CD pipeline for CPS development.

### 2.1 Development of CPSs

CPSs are more complex and difficult to design, develop, test, and integrate than conventional software systems [32, 51, 73, 74]. Specifically, Törngren and Sellgren [74] investigated how CPSs' engineering deals with the complexity of CPS design, and of the environment in which CPSs operate. In this context, it is of paramount importance to perform run-time verification of safety requirements [27], as well as testing encapsulating **Model-in-the-Loop (MiL)** [66], **Software-in-the-Loop (SiL)**, and HiL [2]. With respect to previous studies, we investigate how CPS complexity impacts the setting of CI/CD pipelines, and how developers deal with such complexity.

Considering the costs, risks, and complexity of conducting system testing in a real environment [12, 40], simulation is becoming one of the cornerstones in developing and validating CPSs. CPS developers mainly rely on basic simulation models [29, 67], as well as rigid body [50, 86] and soft body simulation environments [25, 62]. The usage of CPS simulation environments enables automated test generation and execution [37, 54]. However, the limited budget allocated for testing activities and the virtually infinite testing space pose challenges for adequately exercising the CPS behavior [4, 20, 82]. We complement previous studies by looking at the challenges, barriers, and related mitigation strategies when integrating and combining simulators and HiL in CI/CD to support the development, V&V, and evolution of CPSs.

Related to DevOps applications in a CPS context, Park et al. [56] analyzed the use and challenges of the digital twin to enable DevOps approaches for cyber-physical production systems to continuously improve them. Specifically, Park et al. identified challenges related to (i) discrepancies

Table 1. Challenges in CPS Development from Previous Literature

| Ref. | CPS-Related Development Challenges |
|---|---|
| [74] | Environment complexity, co-designing hardware and software |
| [27] | Test generation/automation, verification of safety requirements |
| [2] | Integration of MiL, SiL, and HiL |
| [12, 40] | Where testing is performed (HiL vs. simulators) |
| [25, 29, 50, 62, 67, 86] | Implementation of simulators |
| [4, 20, 82] | Simulator challenges/adequacy |
| [52] | Standards, long build, security, architecture, test environments of embedded systems |
| [56] | Digital twin adoption in manufacturing and related design challenges |

between models and their physical counterparts, (ii) integration between heterogeneous models due to the complexity of CPSs, and (iii) security issues due to the tight coupling between the digital twin and the physical environment. Instead of only looking at automating the production process, we focus more on the CI/CD process for CPS development and evolution.

Finally, Mårtensson et al. [52] identified factors to consider for applying CI to software-intensive embedded systems, such as complexity of user scenarios, compliance to standards, long build times, security, and test environments. These factors represent real impediments for companies that want to adopt CI for embedded systems. Although using different research methods, our study is wider than that of Mårtensson et al. (10 semi-structured interviews, plus an external survey with 55 participants vs. case studies with two companies) and considers the whole CI/CD process from development to delivery to the customer side. Finally, although we confirm findings from Mårtensson et al. [52], our study deepens the analysis of different CI/CD aspects (e.g., setting, phases, and execution environment) for CPSs, and not only in relation to seven CI cornerstones.

Table 1 summarizes the main challenges during CPS development, as stated in previous literature, that are used to drive our study, although we do not focus on specific implementation details of simulators. We leverage the challenges identified by the aforementioned studies to devise the interview guide, particularly those related to (i) the complexity of the underlying environment, (ii) certification and compliance to standards, (iii) test automation, (iv) testing of safety requirements, and (v) MiL, HiL, and simulators.

## 2.2 CI/CD Process

Hilton et al. [34] found that CI is becoming very popular in open source projects. The latter is also true in industry, even if Ståhl and Bosh [69, 70] found that there is not a uniform adoption of CI in industry. Furthermore, Vasilescu et al. [75] showed that CI practices improve developers' productivity without negatively impacting the overall code quality. Finally, Ståhl et al. [71], in a study involving three companies, found that the lack of traceability may prevent the application of CI in conventional software systems.

From a different perspective, Elazhary et al. [18] looked at the extent to which companies follow the CI practices by Fowler and Foemmel [21] through interviews. Their results emphasized differences among companies in terms of repository structure, testing automation, long build, and deployment challenges. Although we share some goals with Elazhary et al., our study, and the dimensions being investigated, relate to CI/CD application for CPS development. In a different study, Elazhary et al. [17] used grounded theory to investigate human factors in CI. Even if our study considers human factors, it is not focused on that.

Vassallo et al. [79] investigated, by surveying developers of a large financial organization, the adoption of the CI/CD pipeline during development activities, confirming what is known from existing literature (e.g., the execution of automated tests to improve the quality of their product), or confuting them (e.g., the usage of refactoring activities during normal development).

Finally, deepening the continuous delivery practice, Chen [10] analyzed 4 years of CD adoption in a multi-billion-euro company, and identified a list of challenges related to CD adoption. Savor et al. [64], instead, by analyzing the CD adoption in two industrial companies, found that it does not negatively impact developer productivity even when the project increases in terms of size and complexity.

Differently from previous studies, our goal is to shed light on the CI/CD process focusing on the peculiarities of CPS development.

### 2.3 CI/CD Barriers and Bad Practices

Different authors studied barriers and/or challenges in adopting CI/CD. These were initially identified by Duvall et al. [13], and are related to the need for maintaining a fully automated build process, handling dependencies, having different levels of builds, and coping with different target environments.

Hilton et al. [33] studied barriers developers encounter when moving toward CI (i.e., quality assurance, security, and flexibility). Olsson et al. [55], instead, looked at the challenges faced while migrating toward CD: the complexity of the deployment environment, the need to achieve timely delivery, and the lack of a complete overview of all the development projects.

Previous research also found that CI/CD may be wrongly applied, leading to bad practices. Specifically, CI/CD antipatterns have been defined by Duvall [15], and empirically elicited by Zampetti et al. [85] from interviews and Stack Overflow posts. Our study is complementary to that. although, where appropriate, we compare the practices observed in our context (CPS specific) with bad practices recommendations from previous studies. Researchers have developed different kinds of tools to detect and remove antipatterns from CI configuration files [23, 78], analyzing the pipeline aging by observing its execution [77], skipping builds [3], or coping with security-related issues in infrastructure-as-code [60].

To the best of our knowledge, there is no such broad investigation on the application of CI/CD in CPS development and evolution, as well as the challenges and barriers faced together with mitigation strategies to overcome them.

## 3 EMPIRICAL STUDY DEFINITION AND PLANNING

The *goal* of this study is to investigate the CI/CD practices for CPS development, to identify challenges and barriers encountered in such practices, together with mitigation strategies adopted to overcome them. The *perspective* is of researchers interested to support developers in configuring CI/CD pipelines for CPSs, and practitioners setting, using, and evolving CI/CD pipelines for CPS development. The *context* from which we have inferred the set of challenges and barriers with related mitigation strategies encountered when setting or evolving CI/CD pipeline for CPS development consists of 10 organizations developing CPSs for eight different domains. To assess the identified set of challenges/barriers and related mitigation strategies, we have surveyed 55 practitioners (not involved in the first step of this study) developing CPSs for nine different domains.

We start by creating organizational profiles by looking at the CI/CD practices adopted by the interviewed organizations, and in general all the practices the organizations are adopting to automate different stages of a build. Specifically, we look at the conditions that determine (i) the setting of the CI/CD pipeline, such as whether an incremental build is used, when the build is triggered, or whether build matrices are used; (ii) the phases instantiated in the pipeline, such as static analysis, various testing levels, or deployment; and (iii) the use of simulators and HiL in the context of the CI/CD pipeline.

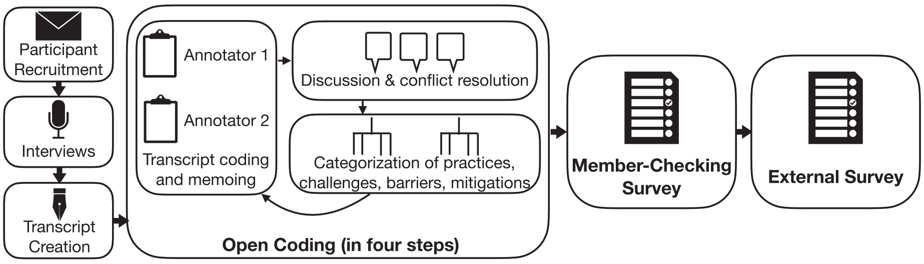
Participant Recruitment
Interviews
Transcript Creation
Annotator 1
Annotator 2
Transcript coding and memoing
Discussion & conflict resolution
Categorization of practices, challenges, barriers, mitigations
Open Coding (in four steps)
Member-Checking Survey
External Survey

Fig. 1. Study methodology.

The study addresses the following two research questions:

- *RQ*$_1$: *What are the challenges and barriers respondents encounter, and how do developers deal with them?* After having characterized the CI/CD and build automation practices, we investigate the challenges (e.g., the need to cope with a slow build or flakiness, or with phases not easy to automate) and barriers (e.g., limited availability of software and/or hardware resources) encountered by the interviewed organizations when dealing with the setting or evolution of the CI/CD pipeline for CPS development. Furthermore, we highlight the strategies (e.g., to adopt a pipeline that relies on both simulators and HiL in different build stages) adopted by the interviewed organizations to deal with challenges and barriers. We validated the relations between challenges/barriers and mitigation strategies through a member-checking survey with the same interviewees or practitioners belonging to the same team of the interviewees involved in the semi-structured interviews.
- *RQ*$_2$: *How relevant are the identified CI/CD challenges/barriers and their mitigation for practitioners involved in CPS development?* In the previous research question, we identified a set of challenges and barriers with related mitigation strategies as experienced by our interviewees; however, this research question aims at performing an external validation by surveying practitioners involved in the setting, evolution, or usage of a CI/CD pipeline for CPS development.

The study methodology used for addressing the research questions is depicted in Figure 1 and described in the following. After having recruited participants be involved in the semi-structured interviews through personal knowledge, we conducted the interviews and transcribed their content. Note that since this is an exploratory study, we prefer to rely on convenience sampling, as previously done in the literature [18, 33]. This is because practitioners involved in CPS development represent a hidden population, and therefore we did not have a sampling frame [8]. The latter helps us conveniently reach a suitable number of study participants. After that, we performed an incremental (in four steps) open coding [35] of the transcripts, discussing the independent coding of multiple annotators, solving conflicts, and creating, through a card sorting strategy [68], categorizations for practices, as well as for barriers, challenges, and mitigation strategies. The relationships between challenges/barriers and mitigation strategies have been validated through a member-checking survey, and finally we performed a further survey to validate our findings beyond the interview context.

## 3.1 Data Collection: Semi-Structured Interviews

We defined the interview structure through an iterative process, which started from the existing knowledge on the topic summarized in Table 1 (see Section 2). From such knowledge, all theoretical pending points were distilled and matched with interview structure areas and questions for each

Table 2. Interview Structure

| Section | Content |
|---|---|
| Overview | Company description, domain, programming languages |
| | Respondent background and role |
| CI/CD pipeline structure | Phases and steps |
| | Tools (versioning, build automation, CI/CD, use of containers) |
| | V&V approaches |
| | Deployment |
| Simulators and HiL | Simulator development/acquisition |
| | Simulator/HiL integration in the pipeline |
| | Simulators vs HiL tradeoffs |
| ML-based components | In the developed software |
| | In the pipeline |
| CI/CD pipeline configuration | Pipeline stability |
| | Build strategies |
| | Triggering |
| Conclusion | Challenges |
| | Barriers |
| | Expected benefits |

interview structure area. As summarized in Table 2, we start with demographics about the organization and the interviewee, and get a first glance at the development and lifecycle management practices [5] adopted in the context of interest. Then, we gather data about the pipeline structure and technology, paying particular attention to V&V and deployment. We then explore the usage of simulators and HiL. We also investigate the presence of any ML-intensive components to be automated (e.g., trained/tuned) or executed by the pipeline over any CPSs' software artifact, or, conversely, the use of ML and Artificial Intelligence (AI) for pipeline automation (e.g., as part of the testing oracle)—that is, AIOps [11]. After that, we investigated how the interviewees configure the overall CI/CD pipeline in terms of build triggering strategies and the possibility to handle different pipeline configurations, each one environment specific. The interview ends with general questions about the main benefits achieved, barriers encountered, and challenges to tackle when configuring and evolving the CI/CD pipeline.

*Interview Participant Selection.* The interview participants have been selected based on personal knowledge, with the goal of identifying experienced practitioners over the theoretical constructs (CI/CD pipelines for CPS) under investigation. The resulting study size (10) is not particularly high, yet it is on the same order of magnitude as similar interview-based studies on CI/CD [18, 33] (although the study by Hilton et al. was followed by a larger survey). It has to be considered also that, differently from previous studies, we targeted a very specific development domain and technology (i.e., application of CI/CD for CPSs in industrial settings). After participants accepted our invitation, we gave them an overview of the questions to expect in the interview, to allow them to gather any additional information.

Table 3 summarizes demographic information about organizations and interviewees involved in the study. Five out of 10 organizations are large (i.e., more than 1,000 employees), 1 is medium (i.e., between 50 and 1,000 employees), 2 are small (between 10 and 20 employees), and 2 are micro (fewer than 10 employees). Furthermore, the sample covers eight different domains: aerospace, automotive, energy, healthcare, railways, robotics, identification technology (i.e., RFID), and acoustic sensors. Finally, the participants' professional experience in the CPS field varies from 3 to 25 years,

Table 3. Participant Demographics

| Org$_{ID}$ | Organization | | Role | CPS |
|---|---|---|---|---|
| | Domain | Size | | Exp. (Y) |
| O$_1$ | Aerospace | Small | R&D Manager | 8 |
| O$_2$ | Healthcare | Large | DevOps Architect | 18 |
| O$_3$ | Acoustic Sensors | Small | SW and HW Integrator | 6 |
| O$_4$ | Robotics | Medium | Team Leader | 7 |
| O$_5$ | Automotive | Large | R&D Manager | 20 |
| O$_6$ | Aerospace | Large | R&D Manager | 20 |
| O$_7$ | Railways | Large | SW and HW Integrator | 10 |
| O$_8$ | Railways | Micro | Team Leader | 25 |
| O$_9$ | Identification Technology | Micro | Software Engineer | 3 |
| O$_{10}$ | Energy | Large | Project Leader | 5 |

with varying job titles, and all of them are currently involved in the configuration of the CI/CD pipeline.

As it will be clearer later, we intentionally selected participants having different maturity levels in the implementation of a CI/CD pipeline for CPSs. In other words, we also included organizations that, although having experience in setting CI/CD pipelines, only partially automated CPS builds, without having a full-fledged CI/CD pipeline. This allowed us to understand, in those cases, how they automated certain phases, as well as the reasons they are still facing challenges in having a complete CI/CD pipeline.

*Conducting Interviews.* Interviews were conducted using a videoconferencing system, by one researcher (with the support of one or two other researchers), following an order based on interviewees' availability. Before starting the interview, the interviewer recalled study goals and gathered consent for recording. The interview structure was followed rigorously, varying only the level of detail over different areas of the interview based on the provided answers. For instance, if a participant mentioned the use of simulators, we asked deeper questions on the topic, whereas we skipped questions not applicable to a given participant. It is important to remark that interviews are treated as independent from each other, meaning that questions were not adjusted over different interviews. This is because, as shown in Table 3, the involved organizations cover a broad range of domains, and the main goal was to achieve a similar understanding among those domains.

*Creating Transcripts.* After interviews have been completed, a researcher transcribed the audio, creating a document organized into sections as shown in Table 2. The transcripts contain a total of 15,329 words and 787 sentences.

## 3.2 Data Analysis from Interview Transcripts

Two authors, experts of the domain (hereinafter referred to as "coders"), independently used online spreadsheets to assign codes (i.e., open coding) to sentences in the transcripts. The coding was carried out following the approach illustrated by Hoover [35] (i.e., annotating a code near sentences of the transcript). A code is defined as a mnemonic label identifying a concept defined in a text fragment (e.g., by applying the label 'TEST' to any part of text reflecting a software testing activity). Wherever appropriate, the coder added a memo that could be leveraged to better explain the observed phenomenon, as well as to identify possible relationships between codes dealing with different aspects of the CI/CD pipeline setting and evolution.

Open coding has been performed over four subsequent sessions by arranging the 10 interview transcripts into four groups. Each group included two, three, four, and one interview, respectively.

After each coding session, the coders held a discussion meeting, in which similar codes created by multiple coders were merged, and conflicts were resolved. After each round, we computed the Krippendorff $\alpha$ [45] to determine the achieved level of agreement. The obtained $\alpha$ values for the four iterations were 0.65 (close to the minimum acceptability of $\alpha = 0.66$ [45]), 0.71, 0.69, and 0.86 (substantial agreement). Starting from the second iteration, the coders could reuse, through a drop-down cell, codes created during previous iterations, or create new ones. Note that to further limit agreement by chance, each code annotation was reviewed during the meetings, not just the disagreements.

During the discussion meetings, broad groups of codes were also defined. For instance, we distinguished codes belonging to the CI/CD pipeline from those related to the development process. In addition, we started grouping codes belonging to different phases of the pipeline, and codes related to challenges, barriers, and mitigation strategies. Such a categorization started during the first discussion meeting and then was refined over the next ones. After the first two sessions of the open coding (after the first session the set of codes was too immature for this purpose), three researchers iteratively produced—by adopting a card sorting strategy [68]—the first version of a mind map grouping codes into categories. Such a mind map has been used as a support to ease the subsequent open coding phases and to evaluate the extent to which non-leaf nodes were saturated. Note that we do not expect a full saturation [63] in this study, due to the high diversity of the considered application domains. The mind map was then refined after each subsequent coding phase.

Overall, we identified a set of 179 codes, which led to the construction of a categorization of codes explaining the phenomenon, organized across 43 high-level categories.

Finally, the two coders performed three iterations over the transcripts, codes, and memos to derive relations between different codes. For instance, it is possible that process constraints (e.g., the need to use a specific type of simulator or tool imposed by the domain, or to adopt certain coding standards) introduce challenges while setting the pipeline (e.g., the need to cope with phases not easy to automate, or slow build and flakiness) that may be addressed by relying on a particular mitigation strategy (e.g., push small changes when using incremental builds). As an example, when talking about flaky behavior experienced in the build process, $O_4$ mentioned: *"of course, we have some retry for network issues,"* whereas *"in case of resources problems we do not have retries, but the pipeline maintainers can open issues aimed at solving the problem."* The outcome of this step consists of 90 relations from 128 sentences. We will present how different codes relate to each other and are spread among different organizations through storytelling.

## 3.3 Member-Checking Survey to Validate Relations Between Challenges/Barriers and Mitigation Strategies

To verify our understanding of how the interviewed organizations act to address the challenges and barriers encountered while setting and maintaining the CI/CD pipeline for CPS development, we conducted a member-checking survey by involving the interviewees themselves, or people working in the same team of the interviewees. Asking outside the same team, especially in large organizations, would have reached completely different projects or even different domains, even unrelated to CPS, making the member-checking worthless.

The survey has been designed by following guidelines for survey design and operation from social science [31] and software engineering [41–44, 57]. Specifically, the survey contains:

- An introduction explaining the study goals;
- A set of 10 sections in which we validate the relations between the 10 challenges/barriers for which we found at least one mitigation strategy from the transcripts; and

- A demographic section in which we asked the participant the application domain, the role within the organization, the years of experience in CPS development, as well as information about the CI/CD pipeline (i.e., (i) whether or not the organization has a CI/CD pipeline in place, (ii) years from its introduction, and (iii) how the participant interacts with it).

Since the main goal of the survey is to validate our correct understanding of the challenges/barriers and related mitigation strategies, we asked the participants to provide their personal contacts (among them the name of the organization) mainly for traceability purposes.

For each section in the survey, we start by asking whether or not the challenge/barrier has been faced at least once by the team they are working with. Specifically, instead of using a yes/no question, we added a third option aimed at highlighting those cases where the challenge/barrier cannot be encountered due to the development process adopted by the organization. For instance, if an organization does not use HiL in its development process, it will never experience problems due to the high cost or lack of scalability of the hardware devices. If the challenge/barrier has been encountered at least once within the organization, we list a set of questions, each one aimed at investigating the adoption of the identified mitigation strategy to overcome the previously presented challenge/barrier. Specifically, the respondent could choose between three different options: (i) yes, and we used it, (ii) yes, but we never used it, and (iii) no. Two out of 17 questions dealing with mitigation strategies provide only two options: (i) yes, it happened, and (ii) no, it never happened. At the end of each section, respondents could use an optional free comment field to provide additional mitigation strategies adopted for overcoming the related challenge/barrier.

The questionnaire was administrated through Survey Hero,[1] and we kept the questionnaire open for 12 weeks. Note that nobody reported having particular issues (e.g., privacy issues) with the used survey administration tool. Furthermore, because of constraints imposed during the survey administration, we had to keep it anonymous.

After closing the survey, we obtained 11 responses from the 10 organizations involved in the semi-structured interviews. Specifically, for $O_5$, we obtained two different responses, even if one of them did not provide demographic information. Among the 10 respondents providing their personal contacts, four of them have also participated in the semi-structured interviews. Furthermore, 4 respondents are R&D managers, 3 are software and hardware integrators, 2 are DevOps architects, and 1 is a DevOps QA engineer. In terms of years of experience with CPS development, 5 respondents have between 1 and 5 years of experience, 2 have between 5 and 10, and the remaining 2 have more than 10 years. Seven out of 9 participants (the ones answering this specific question) declare that their organization already has in place a CI/CD pipeline used while developing CPSs (1 introduced it less than one year ago and 1 has a mature pipeline introduced more than 5 years ago, and 5 between 1 and 5 years ago), and in terms of the way they interact with the pipeline, among the 6 participants who answered this question, 1 only uses the CI/CD pipeline, 2 are involved in its setting and maintaining, and 3 set, maintain, and use it for their development tasks.

## 3.4 Evaluation Through an External Survey

To address $RQ_2$, we conducted a survey involving practitioners using (or trying to set up) a CI/CD pipeline for CPS development in their organization. To recruit participants, we used two different sources:

(1) *Snowball sampling* [30], where we shared the survey link to some personal contacts and encouraged them to indicate us further participants. This choice has been dictated because, although we had a relatively limited set of contacts reachable with our knowledge,

[1]https://www.surveyhero.com/.

snowballing could help us reach the relevant people (those involved in CPS development by relying on a CI/CD pipeline).

(2) *An infrastructure for recruiting survey participants*, namely *Prolific*.[2] This platform allows to reach additional participants by paying a small fee. The platform has a participant screening facility (we required participants to have at least a bachelor's degree in computer science or similar, and knowledge about relevant software development technology, including versioning, monitoring, virtualization, and testing). In addition, similarly to what was done in the member-checking survey, we collected further information about CI/CD competences to further filter participants. At the same time, we are aware that with *Prolific*, we have less control over the participants' reliability than with snowballing. To mitigate this problem, our online package contains separate results belonging to the snowball sub-sample and the *Prolific* sub-sample.

The online survey presented to the participants has (i) an introduction explaining the study goals (i.e., to assess a catalog of challenges and barriers concerning the setting and maintaining of a CI and CD pipeline for CPS development); (ii) 14 sections in which we validate challenges, barriers and mitigation strategies; and (iii) a demographic section similar to the one described in Section 3.3.

We started by asking, for each challenge/barrier (properly grouped in categories), whether they have ever encountered it as a factor preventing/limiting the setting up of a CI/CD pipeline, or, if the participant did not encounter it, whether she perceives the challenge/barrier as a real impediment. Specifically, the respondent could choose between four different options: (i) yes, it is relevant (and I encountered it), (ii) yes, it is relevant (but I never encountered it), (iii) no, I do not consider it as relevant, and (iv) does not apply to my context. If at least one of the challenges/barriers in the category was felt as relevant to the respondent, the survey shows a new section asking about the mitigation strategies used (or felt as relevant) to address the previously selected challenges/barriers by using a multiple choice answer. Specifically, the respondent could choose among the mitigation strategies we obtained as a result of RQ$_1$, but could also add new (unseen) mitigation strategies. Finally, the survey contains an open-ended question aimed at collecting other challenges/barriers that did not apply to the 10 interviewed organizations.

Also in this case, the survey has been administrated through Survey Hero, and nobody reported having particular issues with this administration tool. For the snowball sample, the questionnaire has been left open for 1 month, and due to constraints imposed during the survey administration, we kept it anonymous. For what concerns *Prolific*, we obtained the requested responses within the same days the survey has been opened.

In the end, we obtained 19 responses from the snowball sampling and 50 further responses from *Prolific*. However, through a screening of the participants' answers, we discarded 14 responses from *Prolific*—that is, (i) it was difficult to infer whether or not the participant works for CPS development (e.g., education or applications for cosmetic stores), and (ii) the participant declares to not have a CI/CD pipeline in place within the organization, and at the same time declares that the CI/CD pipeline has been adopted only recently. As a result, we obtain a final set of 55 valid responses covering nine different application domains (as shown in Figure 2).

Among the respondents providing demographic information (51), in terms of the role played in their organization, there are 23 software and hardware integrators, 13 R&D managers, 7 DevOps architects, 5 software developers/testers, 1 project manager, and 1 CTO (chief technology officer). In terms of years of experience with CPS development, 19 respondents have less than 1 year of

[2]https://www.prolific.co.

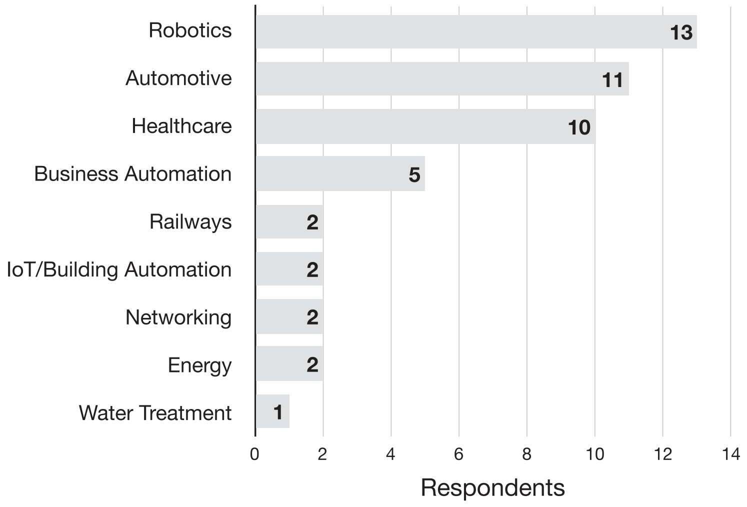

Fig. 2. CPS domains from the external validation survey.

experience, 27 between 1 and 5, 2 between 5 and 10, and the remaining 3 more than 10 years. Forty-seven out of 51 respondents declare that their organization already has in place a CI/CD pipeline used while developing CPS (19 introduced it less than 1 year ago, 7 have a mature pipeline introduced more than 5 years ago, and 21 between 1 and 5 years ago). Finally, in terms of the way our respondents interact with the pipeline, 31 only use the CI/CD pipeline, 6 are involved in its setting and maintaining, and the remaining 10 set, maintain, and use the pipeline for their development tasks. Finally, among the respondents who declare that their organization does not have a CI/CD process in place for CPS development, 3 declare being involved in setting it.

## 4 STUDY RESULTS

In the following, we report and discuss the results addressing the RQs defined in Section 3. To properly contextualize challenges, barriers, and their mitigation strategies, it is important to summarize the development process of the interviewed organizations. Specifically, Section 4.1 briefly describes, for each organization participating in the semi-structured interviews, the CPS development process, focusing more on the adoption of CI/CD pipelines and, in general, on their level of build automation. The interested reader can find more details in the appendix.

### 4.1 Contextualization: Organization Profiles

Table 4 provides an overview of the main analyzed dimensions for the 10 organizations considered in our study. In the following, we briefly describe them.

*4.1.1 $O_1$ (Aerospace).* $O_1$ is involved in V&V tasks for aerospace software (i.e., on-board software for satellites), hence their CI/CD pipeline is only for V&V and not for development. The standards in the aerospace domain enforce the adoption of conventional programming languages (i.e., "*We mainly use ANSI C-99 following the MISRA rules*") as well as the need for certifying software.

$O_1$ started to adopt CI/CD practices for CPSs less than 1 year ago. Due to the application domain and the related standards and certification constraints, the pipeline compiles the software provided and developed by the customer, relies on SonarQube for static code analysis checks, and executes unit and robustness tests to *"check how the system behaves/reacts in the presence of unexpected*

Table 4. Summary of the CPS Development Process Adopted Within the 10 Interviewed Organizations (i.e., $O_{ID}$)

| Property \ Organizations | | $O_1$ | $O_2$ | $O_3$ | $O_4$ | $O_5$ | $O_6$ | $O_7$ | $O_8$ | $O_9$ | $O_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Prog. Language | | C | C# C++ | C C++ | C++ Python | RTJ | C C++ | C | C C++ | C# Java | Java Python |
| Pipeline Maturity | | $< 1$ | $[1, 5)$ | ✗ | — | $< 1$ | — | $[1, 5)$ | ✗ | ✗ | $\geq 5$ |
| Phases | Static Analysis | ● | ● | ✗ | ● | ✗ | ● | ❍ | ❍ | ✗ | ● |
| | Unit Test | ● | ● | ❍ | ● | ● | ● | ● | ❍ | ◗ | ● |
| | Int. Test | ✗ | ● | ❍ | ● | ✗ | ● | ● | ❍ | ◗ | ● |
| | System Test | ✗ | ● | ✗ | ● | ✗ | ◗ | ◗ | ❍ | ❍ | ✗ |
| | Non-Func. Test | ● | ◗ | ❍ | ❍ | ◗ | ❍ | ◗ | ❍ | ◗ | ● |
| | Deploy | ✗ | ● | ◗ | ● | ● | ● | ● | ◗ | ◗ | ● |
| Triggering | Continuous | | ✓ | — | ✓ | ✓ | ✓ | ✓ | — | — | |
| | Incremental | | ✓ | — | | | | | — | — | ✓ |
| | Nightly | ✓ | ✓ | — | ✓ | ✓ | | | — | — | |
| Pipeline Config. | Env. | Stable | Domain specific | — | Stable | — | Device specific | Device specific | — | — | Stable |
| | Staged Builds | ✗ | ✓ | — | ✗ | ✗ | ✗ | ✓ | — | — | ✗ |
| Mocking | | ✗ | ✗ | — | ✗ | ✗ | ✗ | ✗ | — | — | ✓ |
| Simulators | | Ext. | Int. | — | Ext. | Int. | Int. | Int. | Int. | Int. | Int. |
| HiL | | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Containerization | VMs | — | — | — | _ | ✓ | — | ✓ | — | ✗ | ✗ |
| | Docker | — | — | — | Deploy and HiL | ✗ | ✗ | — | — | Deploy | HiL Deploy |

The ✓ (✗) occurs when the property (does not) apply to the organization, the — represents cases where the property is not applicable/available for the organization, ● means that the phase is automatized within the pipeline, ◗ means that the phase is automatized but not included in the pipeline, and ❍ means that the phase is done manually.

*inputs."* The triggering of the pipeline is almost manual, even if there are scheduled nightly builds for running test suites requiring a long time to complete.

Finally, $O_1$ cannot involve HiL in the pipeline, as it would require a clean room not accessible from the outside. Instead, it relies on third-party simulators provided by the customer, reducing the costs/efforts needed to develop the simulators from scratch, as well as guaranteeing the trustworthiness of the outcome being produced.

*4.1.2 $O_2$ (Healthcare).* $O_2$ is a large organization involved in the healthcare domain. It adopts conventional programming languages (i.e., mainly C# and C++ during the development process).

$O_2$ has a CI/CD pipeline in place for CPS development that was introduced 4 years ago, and they are still improving it. Furthermore, based on its application domain, $O_2$ is constrained to *"follow medical application frameworks providing a base set of rules in terms of how to build applications and how to integrate them."*

$O_2$ adopts both incremental and nightly builds. Nightly builds leverage HiL and run three different types of testing, namely unit/component, sub-system, and system testing, whereas incremental builds leverage self-developed simulators to provide developers fast feedback about the impact of their changes (i.e., only a subset of the whole set of functional tests are executed). Furthermore, both incremental and nightly builds run static code analysis tools. Finally, nightly builds imply an automated deployment on a "real" **Computed Tomography (CT)** scanner (i.e., *"physical systems that are equivalent to the real hardware in the CT scanner but not connected to anything around it which has a simulator running on it."*

*4.1.3 $O_3$ (Acoustic Sensors).* $O_3$ is involved in CPS innovation for the industry, among others, the development of the SPL Noise Meter Board, by using conventional programming languages (i.e., Python for testing and C and C++ for micro-controllers development). Each team is composed of both software and hardware experts who work together.

$O_3$ does not have a CI/CD pipeline for CPS development; however, the deployment is fully automated, whereas the testing is manual (i.e., impossibility to automatically test acoustic signals). Finally, at the moment, $O_3$ only uses real hardware devices, yet they wish to include simulators in their CI/CD process.

*4.1.4 $O_4$ (Robotics).* $O_4$ is involved in the development of autonomous robots, and similarly to $O_3$, each team accounts for both hardware and software experts. In their development process, $O_4$ mainly adopts C++, together with Python for users' interfaces and for interacting with the hardware devices.

$O_4$ has a fully containerized (using Docker) pipeline for CPS development. It relies on continuous and nightly builds for running regression testing activities on already packaged components and for deployment to the customers. Furthermore, continuous builds also execute static code analysis tools to inform developers about code quality degradation, and unit tests relying on simulators. The application domain does not introduce certification constraints, whereas it hinders the automation of non-functional testing within the pipeline. Finally, $O_4$ relies on third-party simulators and HiL into different stages of the whole CI/CD process.

*4.1.5 $O_5$ (Automotive).* $O_5$ is a large organization operating in the automotive domain working on the software-focused driving platform. This is the only organization in our study relying on real-time languages (i.e., real-time Java to cope with scheduling requirements of embedded systems).

$O_5$ already has a CI/CD pipeline in place mainly for deployment purposes, even if it is working on improving it. However, unlike the others, $O_5$ relies on virtual machines instead of using Docker containers. Moreover, $O_5$ does not test all the developed modules together since it *"deploy[s] individual bundles to a platform."*

Finally, since $O_5$ develops software for embedded entertainment in the automotive domain, the HiL is only available for a final validation on the customer's side, so most of the work is done relying on virtual environments.

*4.1.6 $O_6$ (Aerospace).* $O_6$ operates in the aerospace domain, and it is mainly involved in developing and refining the routing algorithm for the FRA (Free Route Airspace). Similarly to $O_1$, it relies on conventional languages: *"C and C++ [are] used for the back-end."*

$O_6$ already has a CI/CD pipeline including static code analysis, unit testing, integration testing, and deployment. Similarly to $O_1$, it is required that the developed code satisfies strict certification requirements that are mainly checked by relying on code coverage tools. However, differently from other organizations, $O_6$ does not rely on nightly builds, meaning that also time-intensive tasks are executed at each change: *"even the slow builds are continuously built."* Finally, the pipeline provides a monitoring mechanism for what concerns aspects of the real-time operating system such as scheduling and memory that *"gives us the possibility to collect feedback/evidence that may help us in obtaining the certifications."*

With regard to HiL and simulators, $O_6$ relies on both, however it does *"not have simulators and HiL in the same pipeline mostly for certification issues."*

*4.1.7 $O_7$ (Railways).* $O_7$ is involved in delivering software for railways (i.e., TCMS (Train Control Management System)). In terms of programming languages being used, the interviewee mentions the need of adapting the programming language to the device on which the software has to be executed.

$O_7$ already has a CI/CD pipeline in place for CPS development that, at the moment, is in a continuous improvement state. Based on the application domain, $O_7$ adopts staged builds following the "green-build rule." In the first stage, the build process is executed on a virtual machine, and in the presence of a green status, all the components are deployed together, enabling the execution on the virtual train. In the presence of a green status, it is possible to move to the next stage that relies on the hardware test track, *"where [there is] the whole set of devices and even some more that [are not] in the virtual train."* Finally, in the presence of a green status, it is possible to run the last stage relying on a real train. All the stages include functional testing, whereas the deployment is automated only for the first stage. Based on the previous statements, it is possible to conclude that $O_7$ adopts both simulators and HiL in different stages of the build process, with the use of HiL occurring only in the last stage of the pipeline.

*4.1.8 $O_8$ (Railways).* $O_8$ is involved in the railways domain (i.e., the development of a specific component used for transmitting data between on-board and ground applications). $O_8$ uses C and C++ (i.e., conventional programming languages), and it has strict certification requirements (e.g., compliance with the railway standards and specifications).

$O_8$ does not have a CI/CD pipeline in place for CPS development and it has, in general, little automation in the development process (i.e., only the adherence to standards and specifications is automated). Finally, due to the high cost of the hardware devices in this particular domain, $O_8$ mainly relies on simulators that are self-developed. However, once per week, $O_8$ performs a testing session with a real *"train running in a real environment with real traffic."*

*4.1.9 $O_9$ (Identification Technology).* $O_9$ is involved in *"develop[ing] software relying on identification technologies such as RFID [Radio Frequency IDentification], Bluetooth low energy or bar codes"* relying on conventional programming languages such as Java and C#.

Due to a lack of culture for setting a pipeline dealing with sensors and actuators, $O_9$ does not have a CI/CD pipeline for CPS development. However, the testing phases are almost fully automated. For what concerns the deployment of CPS-related software, $O_9$ relies on Docker for creating images that are manually deployed onto the servers. The development process also features a monitoring component for the internal development platform and customers' devices, to notify about anomalies and errors, as soon as they occur. Finally, the development process considers both (self-developed) simulators and HiL.

*4.1.10 $O_{10}$ (Energy).* $O_{10}$ is involved in the development of prototypes and proof of concepts for the energy domain. It has a mature (i.e., introduced in 2016) pipeline for CPS development that uses conventional programming languages, mostly Java and Python.

Other than having a compilation phase, the CI/CD pipeline is aimed at executing static code analysis tools and linters, unit, and integration tests, followed by a deployment phase where the packaged version of the software is usually stored into an artifact repository as a docker image. $O_{10}$ does not rely on nightly builds; it only uses incremental builds.

$O_{10}$ does not need to run the software on embedded devices, implying that $O_{10}$, other than simulating the hardware when needed, mainly replaces it with mock-ups. Only when the real devices are available and it is safe to use them for testing does $O_{10}$ use Docker images for checking the correct behavior over the real devices.

## 4.2 $RQ_1$: What are the Challenges and Barriers Respondents Encounter, and How Do Developers Deal with Them?

This research question describes barriers and challenges emerging from the semi-structured interviews. We start by describing the challenges related to the CPS development process in

Table 5. Process-Related Challenges

| Category | ID | Challenge | Organizations |
|---|---|---|---|
| General | $PRC_1$ | Cycle-time reduction | $O_2$ |
| | $PRC_2$ | Onboard developers | $O_7$ |
| Culture | $PRC_3$ | Limited CI/CD culture | $O_1$ |
| | $PRC_4$ | Limited CI/CD culture for CPS development | $O_9$ |
| Environment | $PRC_5$ | Complexity of the environment | $O_2$, $O_4$, $O_5$, $O_6$, $O_8$ |
| | $PRC_6$ | Variability of the environment | $O_7$ |
| | $PRC_7$ | Lack of redundancy in the environment | $O_7$ |
| Testing | $PRC_8$ | Test cases manually derived | $O_5$, $O_8$ |
| | $PRC_9$ | Test cases manually executed | $O_3$, $O_9$ |
| | $PRC_{10}$ | Different interpretations for the same requirements | $O_7$ |
| | $PRC_{11}$ | Need a controlled environment for test automation | $O_3$, $O_4$ |
| | $PRC_{12}$ | Complexity in oracle specification for test automation | $O_3$, $O_5$, $O_6$, $O_8$, $O_9$ |
| | $PRC_{13}$ | Complexity for deriving integration tests | $O_{10}$ |
| | $PRC_{14}$ | Complexity for deriving safety tests | $O_4$, $O_5$, $O_8$ |
| Deployment | $PRC_{15}$ | Late deployment | $O_2$ |
| | $PRC_{16}$ | Expensive deployment | $O_7$ |
| Simulators | $PRC_{17}$ | Lack of trustworthiness for simulators | $O_3$ |
| | $PRC_{18}$ | Complexity for oracle automation with simulators | $O_8$ |

general. Then, we describe barriers and challenges encountered when setting and maintaining the CI/CD pipeline for CPS development, together with the related mitigation strategies. Note that we did not find mitigation strategies for all the barriers and pipeline-related challenges, and as described in Section 3.3, the member-checking survey only considers the barriers/challenges for which there was an explicit mitigation strategy reported by at least one of the interviewed organizations.

*4.2.1 Process-Related Challenges.* Table 5 reports the process-related challenges identified in our interviews, together with the traceability among which challenge has been encountered by which organization. It is important to remark that process-related challenges may not be specific to CI/CD, but are, more in general, challenges in the development process that, based on what was reported by the interview participants, have an impact on setting up and maintaining a CI/CD pipeline.

The challenges have been grouped into six different categories: general, culture, environment, testing, deployment, and simulators. For each category, in the following, we provide a brief description of the challenges belonging to it, together with some examples.

*General.* This category accounts for two challenges, each one mentioned by only 1 out of 10 organizations. One of the main benefits of adopting a CI/CD pipeline is related to the overall cycle time reduction ($PRC_1$). However, even if $O_2$ has already invested effort and money in reducing the release time, it already sees space for reducing it: *"The biggest problem . . . is cycle time. Three years ago, the cycle time was six weeks, while now we could do it every day. It is still not enough from a developer perspective because the feedback is not fast enough."* Although this challenge also applies to conventional software, when it comes to the CPS context, the challenge is exacerbated mainly due to the need of interacting with both HiL and simulators. In this regard, $O_2$ mentioned that the cycle time cannot be easily reduced due to (i) the high costs for the infrastructure and (ii) the translation of test strategies to hardware devices being very demanding.

$O_7$ is facing problems when trying to onboard new developers ($PRC_2$) mainly due to the complexity of the railways' domain, as also found by Törngren and Sellgren [74]. The interviewee stressed that in the railways' domain, it is crucial to follow specific standards that need to be known and properly understood by developers and testers.

*Culture.* This category groups two challenges related to the presence of a limited CI/CD culture in the development teams. This may limit the possibility of properly leveraging CI/CD facilities throughout the development process. $O_1$ reports the adoption of a pipeline that only includes tasks that are easy to automate mainly due to *"lack of knowledge"* ($PRC_3$), as also found by Zampetti et al. [85]. Instead, although $O_9$ has already in place a pipeline for developing and deploying mobile apps to the app-store (i.e., *"The setting of a CI/CD pipeline in the mobile context has been very easy"*), it does not have a pipeline for CPS development due to *"a lack of a deeper knowledge in the CI/CD context for CPS,"* in particular for what concerns the interaction between software and hardware components ($PRC_4$). Specifically, there is a need for knowledge on how to properly account for the inclusion and setting of both HiL and simulators in the CI/CD pipeline configuration, as well as how to include a feedback mechanism to gather information directly from the field.

*Environment.* This category features three different challenges dealing with the characteristics of the physical environment in which the developed code has to be deployed.

Among them, only $PRC_5$ (i.e., environment complexity) is mentioned by multiple organizations (5 out of 10), whereas the remaining 2 only come from $O_7$. The complexity of the environment impacts the execution environment being set (i.e., simulators or HiL). The unavailability of third-party simulators (and the need for self-developing them) impacts the ability to simulate certain behaviors, or even in deviations between HiL and simulated environments. The consequence is that builds executed on simulators will have a different outcome when run on HiL. For instance, $O_4$ mentioned: *"Walking is not so easy to simulate so we need a real walking robot for spotting bugs,"* whereas $O_8$ stated: *"It could be difficult, demanding and expensive to have a one-to-one relationship between simulators and real systems."* Our findings stress what is already known from previous literature in terms of relying on simulated environments—that is, the testing over simulators may fail to expose problems that would only manifest when running the system on the real hardware [52].

$O_7$ faces a problem related to the high environment variability ($PRC_6$) [74], due to trains having different characteristics: *"We can rarely copy-paste software that has to run on different train architectures."* At the same time, $O_7$ also faces a challenge due to the structure of its development process that is not cloud-based and has no redundancy ($PRC_7$), implying that *"in the presence of network issues or server issues we are totally black and this is affecting everyone."*

*Testing.* This category groups seven challenges. $O_5$ and $O_8$ mention as a challenge the substantial manual effort required for the test case specification process ($PRC_8$). $O_3$ and $O_9$, instead, felt the manual execution of testing activities to be challenging—that is, $PRC_9$ (e.g., *"Another big barrier is related to the test case execution that, at the moment, we are doing manually since both the environment setting and the oracle definition require manual intervention"* for $O_9$). Our findings confirm what is already pointed out by Mårtensson et al. [52] in terms of the presence of complex user scenarios implying the need of manual testing.

$O_7$ found it difficult to automate the test case specification mainly because the standards might be interpreted differently by different developers, and both might be correct ($PRC_{10}$): *"how do you read the standard? The standard is interpreted so the same requirement can be differently interpreted by different people (a challenge for automation)."* A different challenge experienced by $O_3$ and $O_4$ is related to the need for a controlled test environment ($PRC_{11}$) impacting the execution environment to be used in the pipeline. For instance, $O_3$ mentioned: *"Since the output of the system is sound and the test should check the sound quality it is better to have it in a controlled environment that makes use of simulation."*

Another test automation challenge is related to oracle specification ($PRC_{12}$), as mentioned by 5 out of 10 organizations. The impossibility of specifying an automated oracle hinders what kinds of tests one can run in the pipeline. This may happen, for instance, when one needs to evaluate a signal received from a sensor—that is, *"The main challenges for automatizing the test execution: a good way to model the test itself and have an oracle that can compare with the actual behavior"* stated by $O_3$. This aspect has already been mentioned by Mårtensson et al. [52]; however, although they only talked about usability testing, we stress more the impediment in automatically determining and checking the test oracles, also for functional testing mainly due to outcome coming from real hardware devices working in a real environment with many external factors to control for (e.g., to check the quality of the acoustic signal coming from sensors) ($O_3$).

The remaining two challenges are related to difficulties encountered when specifying/deriving integration ($PRC_{13}$) and safety ($PRC_{14}$) tests. With regard to the former, $O_{10}$ develops prototypes requiring the interconnection of many different sub-components. This makes it difficult to determine the expected system behavior: *"It is quite hard to derive integration test cases due to the complex combination of all different parts."* With regard to the specification of safety tests, in agreement to what indicated by Gautham et al. [27], $O_4$, $O_5$, and $O_8$ pointed out the complexity to identify situations *"that could never happen"* or *"that you do not expect to happen."* Checking for safety requirements is highly important, especially in those domains, such as aerospace and railways, where the safety integrity level of the system must be equal to or higher than 3.

*Deployment.* This category features two challenges occurring when deploying software on the customers' side. Having deployment too late in the development process ($PRC_{15}$) may result in installation issues ($PC_9$ shown later in Table 7), as experienced by $O_2$: *"we will not be able to run the software on the system because the installation even does not work on the system, because the update/upgrade does not work, or because the system behavior is not being considered in the early stages of development."*

Then, there are cases where the deployment is expensive ($PRC_{16}$) in terms of time and effort needed to complete it. This impacts both the type of execution environment adopted within the pipeline, as well as the build triggering strategy. As experienced by $O_8$ in the railways' domain, the deployment on a test track requires *"one day with people involved in the testing and on a train a couple of days where many people need to be involved."*

The late and expensive deployment is strictly related to the CPS nature. Indeed, as already highlighted in Section 4.1, the organizations deploy on real hardware devices only during the last stages of the overall CI/CD process, mainly due to the high costs of the hardware in specific domains such as railways and aerospace.

*Simulators.* The last category, among the process-related challenges, deals with the usage of simulators. $O_3$ pointed out the presence of scenarios where it is complex to trust the outcome provided by the simulators since there might be many external factors impacting the behavior of the system in a real environment ($PRC_{17}$). Finally, as reported by $O_8$, some scenarios cannot rely on simulators. Specifically, if it is complex for a human to specify the expected behavior for some scenarios, of course, it is not possible to rely on simulators that can emulate the same behavior ($PRC_{18}$).

*4.2.2 Barriers for CI/CD Pipeline Setting and Maintaining and Related Mitigation.* Table 6 summarizes the five barriers encountered by the 10 organizations when applying CI/CD to CPSs. These barriers have been grouped into two categories, described in the following.

*Resources.* This category groups the barriers dealing with limited availability of human ($B_1$) and software and/or hardware resources ($B_2$), both influencing the type of execution environment

Table 6. Pipeline-Related Barriers

| Category | ID | Barrier | Organizations |
|---|---|---|---|
| Resources | $B_1$ | Limited human resources | $O_8$, $O_9$ |
| | $B_2$ | Limited availability of software and/or hardware resources | $O_1$, $O_2$, $O_3$, $O_4$, $O_5$, $O_6$, $O_7$, $O_8$, $O_9$, $O_{10}$ |
| Domain | $B_3$ | Complex non-functional requirements | $O_6$ |
| | $B_4$ | Security configuration prevents CD | $O_2$ |
| | $B_5$ | HiL not usable (e.g., for safety or security reasons) | $O_1$, $O_2$, $O_8$, $O_9$, $O_{10}$ |

adopted within the pipeline. Although we are aware that those barriers can also apply to conventional software systems, the barriers worsen for CPS development, where it is mandatory (i) to rely on simulators, mostly self-developed where you need high expertise about the domain, and (ii) to use HiL that is very expensive particularly in the CPS domain, such as railways and aerospace. For instance, $O_8$ mostly relies on HiL due to limited availability of human resources having the skills needed to develop/configure simulators: *"given the needs and the budget of our company, it's much better for more complex scenarios to rely on the hardware in the loop and only use simulations when whatever needs to be simulated is very simple."*

All the interviewed organizations reported the limited availability of software and hardware resources. Specifically, $O_6$ mentioned: *"Based on the fact that in the avionics domain the cost of the hardware is very expensive, we do most of the work in simulated environments,"* whereas $O_7$ stated that *"Resources for the hardware devices (hardware test tracks and testbeds as real trains) represent an issue for us. We have a limited number of test tracks."*

As reported later in Table 8, the analysis of the interviews' transcripts has elicited two mitigation strategies: (i) prioritize and select the test cases to be included within the pipeline (i.e., *"Some strategies rely on genetic algorithms to optimize the resources available for the testing execution environment"* from $O_1$), and (ii) adopt incremental builds mainly relying on impact analysis, as reported by $O_2$: *"for what concerns rolling builds we try to limit the amount of testing being executed in them to be as fast as possible."* The member-checking survey confirms the previous findings, and, as shown later in Table 8, 6 out of 10 organizations ($O_1$, $O_2$, $O_5$, $O_6$, $O_7$, $O_{10}$) report to rely on test prioritization, whereas $O_3$, $O_4$, $O_8$, and $O_9$ consider it useful while having never used it. With regard to the adoption of incremental builds, instead, $O_1$, $O_2$, $O_4$, $O_7$, and $O_9$ mention its adoption, whereas $O_8$ considers it a useful approach to deal with limited hardware/software resources.

Alternative solutions reported in the member-checking survey to cope with limited availability of resources are *"architectural changes with improved testing concepts"* ($O_7$) and, unsurprisingly, *"platform virtualization"* ($O_5$).

*Domain.* This category includes three different barriers, two of them highlighted by only one organization. Specifically, $B_3$ and $B_4$ are related to difficulties arising when automating certain phases in the CI/CD pipeline. For instance, $O_6$ had to cope with the use of a real-time operating system that made task automation difficult: *"the complexity of integrating within the pipeline the execution of nonfunctional testing and system testing,"* whereas $O_2$ could not implement automated deployment due to security policies for the healthcare domain: *"We cannot deploy at the moment because a change in the security configuration of the software prevented our standard [deployment] process."*

$B_5$ is related to coping with a complex execution environment. Specifically, $O_{10}$ mentions that they could not integrate HiL in the CI/CD pipeline for safety reasons and adopts simulation/mocking for the hardware devices to overcome it. As shown later in Table 8, all the organizations facing this barrier used the same mitigation strategy to deal with it. Furthermore, $O_2$ mentions the possibility to rely on *"digital twin hardware that avoids the safety issues (no moving parts, no radiation) but simulates the hardware to some much better."*

Table 7. Pipeline-Related Challenges

| Category | ID | Challenge | Organizations |
|---|---|---|---|
| Pipeline Properties | $PC_1$ | Long build execution time | $O_1$, $O_2$, $O_4$, $O_5$, $O_6$, $O_7$, $O_8$ |
| | $PC_2$ | Build time estimation | $O_9$ |
| | $PC_3$ | Static code analysis tools configuration | $O_3$, $O_7$ |
| | $PC_4$ | Lack to access the production code from the pipeline | $O_1$ |
| | $PC_5$ | CI/CD configuration highly coupled with the environment | $O_2$, $O_5$ |
| | $PC_6$ | Reusability of build artifacts | $O_2$ |
| Thoroughness | $PC_7$ | Development environment detached from the execution environment | $O_1$ |
| | $PC_8$ | Detecting deployment-related errors | $O_2$, $O_6$ |
| | $PC_9$ | Continuous installation | $O_2$, $O_3$, $O_4$, $O_5$, $O_7$, $O_8$ |
| | $PC_{10}$ | Closing the loop introduces performance degradation | $O_5$ |
| | $PC_{11}$ | Complexity in closing the loop due to uncontrollable factors | $O_4$, $O_9$ |
| | $PC_{12}$ | Complexity in closing the loop due to data collection from the field | $O_5$ |
| Simulators | $PC_{13}$ | Limited in their functionality | $O_1$, $O_2$, $O_4$, $O_5$, $O_7$, $O_8$, $O_9$, $O_{10}$ |
| | $PC_{14}$ | Functional correctness | $O_5$, $O_6$, $O_7$, $O_{10}$ |
| | $PC_{15}$ | Deal with real-time properties | $O_5$, $O_9$ |
| | $PC_{16}$ | Interaction with the environment | $O_2$, $O_3$, $O_4$, $O_5$, $O_6$, $O_7$, $O_8$, $O_9$ |
| | $PC_{17}$ | Accessibility | $O_1$, $O_5$, $O_7$, $O_9$, $O_{10}$ |
| HiL | $PC_{18}$ | Availability | $O_{10}$ |
| | $PC_{19}$ | Automated deployment on HiL | $O_7$, $O_8$, $O_9$ |
| | $PC_{20}$ | Test automation on HiL | $O_2$, $O_4$, $O_6$, $O_7$, $O_9$ |
| | $PC_{21}$ | Costs and scalability | $O_1$, $O_2$, $O_3$, $O_4$, $O_5$, $O_7$, $O_8$, $O_9$ |
| Flaky Behavior | $PC_{22}$ | Dependency installation | $O_4$ |
| | $PC_{23}$ | Features' interaction | $O_2$ |
| | $PC_{24}$ | HiL availability | $O_{10}$ |
| | $PC_{25}$ | HiL inputs | $O_5$, $O_{10}$ |
| | $PC_{26}$ | Lack of control over resources | $O_2$, $O_4$, $O_5$, $O_6$, $O_7$, $O_9$, $O_{10}$ |
| | $PC_{27}$ | Network issues | $O_1$, $O_2$, $O_4$, $O_5$, $O_6$, $O_7$, $O_9$, $O_{10}$ |
| | $PC_{28}$ | Timing issues | $O_4$, $O_{10}$ |

*4.2.3 Pipeline-Related Challenges and Related Mitigation.* Table 7 summarizes the pipeline-related challenges faced by the 10 organizations. The challenges have been grouped into five categories, each one related to a specific aspect of the CI/CD pipeline setting and evolution: pipeline properties, thoroughness, simulators, HiL, and flaky behavior. In the following, we discuss each identified challenge, together with some examples from the study participants' experiences, and related mitigation strategies.

*Pipeline Properties.* This category accounts for six different challenges, two of which deal with the build execution time ($PC_1$ and $PC_2$), whereas the remaining four are related to the overall pipeline configuration. Four out of 10 organizations faced long build execution time, influencing the type of tasks automatized within the pipeline. For example, $O_6$ mentioned: *"Slow builds hinder the inclusion of running non-functional testing in the pipeline."* Although this is also considered a relevant challenge for conventional applications [14, 77, 85], for CPSs the problem can be further exacerbated when deploying and executing software on simulators or HiL. The latter confirms what is already found by Mårtensson et al. [52] highlighting how working with a highly integrated (tightly coupled) system, a small delivery to the main track may cause building and linking of a large part of the system resulting in long build times. The latter has been also mentioned by $O_2$ where there is a single integration branch where the components developed by their 70 teams are integrated into a single join point: *"each component has a test service so running unit tests is very fast but we have a huge amount of high-level testing that is easy to write but kills us in terms of execution time."* By looking at the result of the survey (Table 8), the interviewed organizations mentioned a wide set of actions to deal with the preceding challenge. One possibility is to prioritize and select only a subset of test cases in the test suite to be executed (used also by $O_1$, $O_2$, and $O_7$, and considered a useful action by $O_4$ and $O_8$). A different approach, highlighted by $O_2$, deals with the introduction of parallelization within the overall build process: *"We have 20 test machines in parallel for*

Table 8. Relations Between Challenges/Barriers and Mitigation Strategies as Seen from the Semi-Structured Interviews and the Member-Checking Survey

| **Challenge/Barrier** | **Mitigation** | **Organizations** |
|---|---|---|
| $B_2$: Limited hw/sw resources | Test Prioritization | $O_1$, $O_2$, $\mathbf{O}_3$, $\mathbf{O}_4$, $O_5$, $O_6$, $O_7$, $\mathbf{O}_8$, $\mathbf{O}_9$, $O_{10}$ |
| | Incremental Builds | $O_1$, $O_2$, $O_4$, $O_7$, $\mathbf{O}_8$, $O_9$ |
| $B_5$: Domain hinders HiL | Rely on sim./mock-up | $O_1$, $O_2$, $O_8$, $O_9$, $\mathbf{O}_{10}$ |
| $PC_1$: Long build | Test Prioritization | $O_1$, $O_2$, $\mathbf{O}_4$, $O_7$, $\mathbf{O}_8$ |
| | Adopt Parallelization | $O_2$, $O_4$, $O_5$, $O_6$, $O_7$, $\mathbf{O}_8$ |
| | Nightly Builds | $O_1$, $O_2$, $O_4$, $O_5$, $O_6$, $O_7$, $O_8$ |
| | Incremental Builds | $O_2$, $\mathbf{O}_4$, $O_5$, $\mathbf{O}_6$, $O_7$, $O_8$ |
| $PC_9$: Continuous installation | Containerization | $\mathbf{O}_3$, $O_4$, $O_5$, $\mathbf{O}_7$, $\mathbf{O}_8$ |
| $PC_{13}$: Sim. limited func. | Combine sim. and HiL | $O_4$, $O_7$, $O_8$, $O_9$, $O_{10}$ |
| $PC_{16}$: Sim. coupled with env. | Combine sim. and HiL | $O_2$, $O_3$, $O_4$, $O_8$, $O_9$ |
| $PC_{17}$: Sim. accessibility | Timeout | $O_1$, $O_5$, $\mathbf{O}_7$, $\mathbf{O}_{10}$ |
| $PC_{21}$: HiL costs and scalability | Combine sim. and HiL | $\mathbf{O}_1$, $O_2$, $\mathbf{O}_3$, $O_4$, $\mathbf{O}_5$, $O_7$, $\mathbf{O}_8$, $O_9$ |
| | Green-build rule | $O_2$, $O_3$, $O_4$, $O_7$ |
| $PC_{26}$: No resources' control | Fix the code | $O_4$, $O_6$, $O_7$, $O_9$, $O_{10}$ |
| | Fix pipeline config. | $O_6$, $O_7$, $O_9$ |
| $PC_{27}$: Network issues | Retry | $O_2$, $O_4$, $O_5$, $O_6$, $O_7$, $O_{10}$ |

Organizations that do not rely on the mitigation are shown in bold but consider it a useful solution.

*managing the overall test size, especially for nightly builds."* The latter is also used by $O_4$, $O_5$, $O_6$, and $O_7$, whereas $O_8$ only felt it as useful. It is also possible to run the whole build process only within nightly builds, even if this may be controversial since it defeats the CI/CD purpose [13]. However, this is considered acceptable for $O_1$, as its pipeline is limited in scope (i.e., used only for V&V purposes). In addition, $O_2$, $O_5$, $O_6$, and $O_7$ rely on nightly builds to execute time-intensive tasks while adopting incremental builds during working hours ($O_2$, $O_5$, and $O_{10}$). The latter is also used by $O_7$ and $O_8$, whereas $O_4$ and $O_6$ consider the mitigation useful even if they have never adopted it.

A different challenge, experienced by $O_9$, that can also apply to conventional systems, although it is more critical for CPSs, is related to the build time variability ($PC_2$), due to the adopted infrastructure *"since our platform works in the cloud we need to know how much time it is required to acquire and elaborate a huge amount of data points."*

Moving to the overall pipeline configuration, in the absence of clear coding standards or guidelines, the adoption of code style checking tools becomes problematic, if not unfeasible ($PC_3$). In this scenario, approaches for coding style inference may be desirable [61, 83]. Similar considerations apply to bug-finding tools, sometimes inapplicable to CPSs for automating code review, as experienced by $O_7$: "*we need expertise on the developers' side for determining whether or not a train is behaving in the expected way.*" The latter is strictly related to $PRC_2$ where, in the presence of safety-critical systems, like the ones in the aerospace and railways domains, it is very difficult to find skilled experts in the domain from both the hardware and software viewpoints.

The lack of access to production code (as experienced by $O_1$) limits the ability to properly set static analysis or testing tools ($PC_4$): *"One big challenge is that we need to guarantee the protection* *of the source code: How to test a component without having its production code?"* The latter is a specialization of the restricted access to information due to security aspects impediment found by Mårtensson et al. [52]. On the same line, there is a challenge ($PC_5$) related to the extent to which technology restrictions, or restrictions coming from the application domain, may impact the pipeline setting. For instance, $O_2$ mentioned that *"the Windows situation does not help us with dockerization,"* and at the same time, they are having trouble in properly configuring the CI/CD pipeline for CPS since *"[they] need to follow medical application frameworks providing a base set of rules in terms of how to build applications and how to integrate them."* The latter results in the last challenge related to the impossibility to reuse previously built artifacts ($PC_6$) in the integration branch (i.e., $O_2$ mentioned: *"It's a huge pain that we do not reuse artifacts"*), mainly due to constraints imposed by the domain.

*Thoroughness.* This category groups six challenges related to (i) ensuring the overall accuracy and completeness of the CI/CD pipeline ($PC_7$, $PC_8$, $PC_9$) scattered across eight organizations, and (ii) closing the DevOps loop by gathering data from the hardware (i.e., $PC_{10}$, $PC_{11}$, and $PC_{12}$ experienced by 3 out of 10 organizations).

$O_1$ faces a challenge related to having a development environment detached from the execution environment ($PC_7$). Another challenge ($PC_8$ experienced by $O_2$ and $O_6$) occurs in the presence of incremental deployment, which makes it difficult to detect and isolate deployment errors. Furthermore, $O_6$ reported how this even makes it necessary to reconfigure the entire pipeline: *"you deploy blocks, if there is an error in one of the blocks detecting it and reconfigure and reset the pipeline is a problem."* Finally, continuous installation ($PC_9$) cannot be achieved due to the late deployment strategy ($PRC_{17}$). This is because changes to the environment impact the pipeline configuration, which needs to be adapted every time. For what concerns continuous installation problems, $O_2$, $O_3$, $O_4$, $O_5$, $O_7$, and $O_8$ have encountered them, with $O_4$ pointing out that by using containerization it is possible to facilitate the switching between software versions to deploy, meaning that it will be possible to handle the variability of the environment in terms of dependencies. As shown in Table 8, containerization is also used by $O_5$, whereas $O_3$, $O_7$, and $O_8$ consider it a viable solution.

Moving on to the need for closing the DevOps loop, the interviews indicated three different challenges hindering the acquisition of data from the physical environment (or hardware device). Working in a CPS context implies having a tight interaction with multiple hardware devices (i.e., sensors and actuators), in which gathering data from them could be problematic due to the presence of many external environmental factors that must be taken into account, as well as the need for having invasive measurement instruments directly in the field. Specifically, $O_5$ stressed the introduction of performance degradation ($PC_{10}$) due to invasive measurement instruments: *"The challenge is that monitoring becomes invasive with respect to the system performance,"* as well as the presence of noise in the collected data ($PC_{12}$): *"There are architectural ways to deal with that so that if some sensor does not update on time, you still can make a relatively informed decision. But even then, you have to make sure that the drift is not over a certain size because then you cannot make reasonable decisions anymore."* $O_4$ and $O_9$ highlighted the presence of uncontrollable factors in a CPS execution environment, making it challenging to close the DevOps loop. For instance, $O_4$ reported: *"Differently from other software applications, there is data that we cannot control such as the presence of something on the floor that the robot is not able to perceive so it will fail. You have to analyze the video data and this is very hard."*

*Simulators.* This category groups five challenges related to simulators' issues and limitations stressed more in the CPS domain due to the high environment complexity [74], which very often results in having scenarios that cannot be emulated, such as in the presence of many external environmental factors to be controlled. Specifically, the need to develop them in-house or the lack of specific skills may lead to simulators that are limited in their functionality ($PC_{13}$). For instance, $O_8$ stated, *"we prefer to spend time in testing on real hardware instead of spending time in developing complex simulators,"* whereas $O_4$ reported, *"Walking is not so easy to simulate, so we need a real walking robot for spotting bugs."* As shown in Table 8, it is a common habit to adopt a pipeline that relies on both simulators and HiL in different build stages to overcome the preceding challenge. A clear example of this happens in $O_7$, where there is a build process made up of three different build stages, each one adopting a specific execution environment (see Section 4.2).

A lack of knowledge about the device/system to simulate can lead to wrong assumptions, affecting the simulator's correctness ($PC_{14}$) as experienced within $O_{10}$: *"This happens more at the beginning of a project when you are not too familiar with the device and you make assumptions on how it works."* These problems might have an impact on the whole CI/CD pipeline setting and trustworthiness, because it is possible to have deviations of the monitored system behavior between the real hardware and simulators.

As experienced by $O_5$, the limited capability to simulate real-time properties ($PC_{15}$) hinders the applicability of simulators or at least raises the need for further tests on HiL. The latter is also confirmed by $O_9$: *"for what concerns the simulation for the RFID we think that the simulation will not give us any benefits due to their unpredictable behavior."*

Likewise for $PC_{13}$, the high level of interaction between different components ($PC_{16}$) forces organizations to directly test feature interaction by using real devices instead of simulating them. Indeed, when using simulators for CPSs, it is important to remark that they have to interact with a too complex environment that must be simulated as well. As an example, $O_6$ mentions problems faced when simulating a car behavior: *"for the CAN data, what do you want to wish to happen here? If you are driving around something you need to know how fast the wheels are turning, as well as what the engine revolutions are together with other sensitive data you might pick up over the canvas. There are a lot of details that are very application dependent."* Also in this case, as shown in Table 8, organizations rely on pipeline configurations including different execution environments—that is, five out of eight organizations facing the challenge declare that this is a useful mitigation strategy ($O_2$, $O_3$, $O_4$, $O_8$, $O_9$).

If an organization has to test third-party software, as in the case of $O_1$, there may be the need to run the simulated environment on a remote machine, which may be problematic when attempting to properly integrated it into a local pipeline ($PC_{17}$), due to network security restrictions. Such a scenario typically occurs in the development of safety-critical systems (which very often are CPSs), because the software needs to be tested by somebody different from the development organization. To deal with this problem, $O_1$ mentions the usage of "timeout" within the pipeline. As shown in Table 8, $O_1$ and $O_5$ handle external simulator unavailability through timeouts, whereas $O_7$ and $O_{10}$ consider this useful yet they do not use it. $O_1$ also mentions they often *"request some customization at the customer side of their simulators. Sometimes it is accepted, most of the times not."*

*HiL.* This category groups four challenges related to issues and limitations of using HiL in the CI/CD pipeline. As shown in Table 7, three out of four challenges in this category are experienced by multiple organizations, whereas $PC_{18}$ is organization dependent. Specifically, $O_{10}$ faces problems with checking hardware availability before running tests ($PC_{18}$): *"One of the biggest problems, when any particular hardware is involved, is that the hardware may either not be available, or it may be switched off."*

From a different perspective, as experienced by $O_7$, $O_8$, and $O_9$, deployment on HiL may be challenging ($PC_{19}$). Specifically, in $O_8$, *"remote installation cannot be used with real systems,"* whereas in $O_9$, *"The other challenge is related to having a fully automated deployment over the customers' server in which it is possible to have full control on what is going on and try to identify, as soon as possible, failures/errors occurring during the deployment."*

Testing on HiL ($PC_{20}$) is considered very demanding to achieve. $O_2$ reports, *"If you translate test strategies to the hardware it is very demanding,"* and this is mostly a consequence of limited human resources being available. However, there are cases where testing on HiL is constrained by the high cost and lack of scalability ($PC_{21}$) of the hardware devices/systems: *"This costs and does not scale"* for $O_2$, or *" it is very costly to test on trains"* for $O_7$.

As shown in Table 8, the study participants identified two possible strategies to deal with these cost and scalability problems: (i) relying on a mixed pipeline where continuous builds run on simulators and some periodic builds on HiL (used by $O_2$, $O_4$, and $O_9$, and considered useful by $O_1$, $O_3$, $O_5$, and $O_8$), or (ii) adopting the green build rule when transitioning between simulators and HiL [15], as highlighted by $O_7$, *"Only when the tests in the virtual train are green can we move to the next step,"* and also used by $O_2$, $O_3$, and $O_4$. The alternative would be, as pointed out by the $O_5$ survey respondent, *"working with virtual devices instead of real hardware devices."*

*Flaky Behavior.* This category accounts for seven different root causes that may lead to non-determinism in the build execution used for CPS development. Flakiness related to non-determinism during test execution [89] has been largely studied [16, 46, 48, 58, 90], and approaches to detect and cope with it have been proposed [47, 49, 59, 65, 87]. Although similar to conventional software, dependency installation within the pipeline ($PC_{22}$) may result in pipelines having a flaky behavior (e.g., for $O_4$, *"ROS uses GitHub repositories for dependency resolution so when GitHub or the repositories are down our build jobs will fail due to the impossibility of resolving dependencies"*) or else little control over external resources ($PC_{26}$) (e.g., *"the most important root cause we experienced is related to the load on the server-side"*)—the root causes behind flaky behavior in CPSs may be different from conventional software. Specifically, a CI/CD pipeline for CPSs can suffer from flakiness due to the following:

- The complex interacting environment ($PC_{23}$) (i.e., CPSs are systems of systems with tight interactions among different components). For example, for $O_2$, *"the complexity of [the] subsystems whose features interact across many indirections may lead to non-deterministic behaviors."*
- HiL unavailability ($PC_{24}$), where without a proper check of the availability of hardware, the build outcome might fail intermittently since the pipeline was not able to properly communicate with the device. $O_{10}$ reported: *"We experienced flakiness in terms of non-deterministic behavior mainly due to hardware not being available."* In this specific scenario, it is important to properly discriminate between intermittent failures caused by communication issues with the HiL from failures due to wrongly implemented functionality.
- Presence of noise in the measurements ($PC_{25}$) when using HiL (i.e., difficulty in removing the effect of external environmental factors from the data read from the sensors, as experienced by $O_5$ and $O_{10}$). Specifically, for $O_{10}$, *"Other times the charge level that you read out would go a little bit higher or there is noise in the measurements,"* whereas for $O_5$, *"you need to understand what your sensors are sensing and what the acceptable range of inputs are."*
- Network issues ($PC_{27}$) where, for instance, glitches in the network lead to a connections being lost, as reported by $O_{10}$, stressing more in the CPS domain where you need to control among the communication occurring across a huge number of different hardware devices operating in a complex environment.
- Simulators not coping with timing issues ($PC_{28}$). For example, $O_{10}$ stated: *"the last problem is related to multi-threaded programming."*

For what concerns flakiness mitigation, as highlighted in Table 8, when the problem is related to the lack of control over resources ($PC_{26}$), the solutions adopted are (i) to change and fix the pipeline configuration (i.e., $O_7$ stated, *"The misbehavior is reported back to the integration team responsible for the Jenkins configuration to find a solution"*), as well as (ii) to fix the root cause of

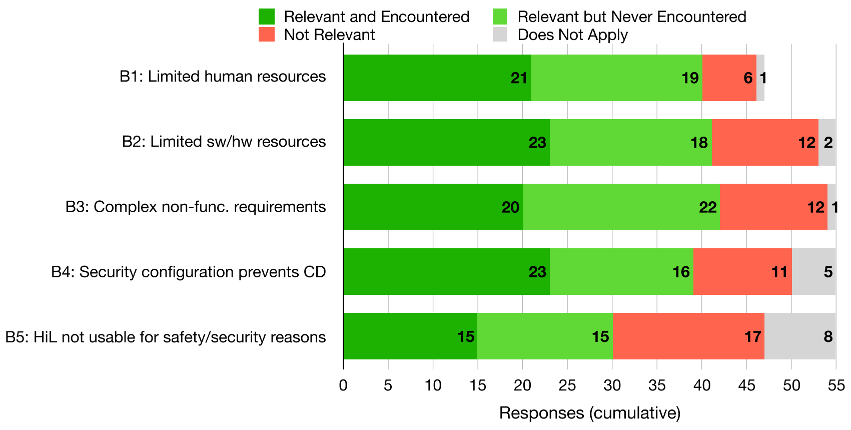

Fig. 3. Results of barriers perception.

the flaky behavior within the code: *"to not experience it anymore in the system"* from $O_2$. When the root cause of the flaky behavior is in the networking ($PC_{27}$), the organizations leverage the "usual" retries ($O_2$, $O_4$, $O_5$, $O_7$, $O_{10}$)—for example, *"of course we have some retry for network issues"* for $O_4$, or *"For what concerns flaky connections, you have to be concerned about missed messages and retries"* for $O_5$. $O_6$ instead only considers it a viable solution. Furthermore, the respondent belonging to $O_2$ mentioned as an alternative solution the *"introduction of quarantine builds together with an appropriate process of how to deal with these tests."*

### 4.3 RQ$_2$: How Relevant are the Identified CI/CD Challenges/barriers and their Mitigation for Practitioners Involved in CPS Development?

This research question describes the results of the evaluation of the findings in RQ$_1$ made through an external survey leveraging practitioners who have not been involved in the semi-structured interviews. Note that we have only validated the barriers and the pipeline-related challenges together with their associated mitigation strategies.

With regard to the five barriers encountered when trying to configure a CI/CD pipeline for CPS development, by looking at the results in Figure 3, we found that among the participants who answered each question, the limited number of human and software/hardware resources together with the presence of complex non-functional requirements to be checked within the pipeline are the ones felt as more relevant (>72%). Furthermore, although 30 out of 55 respondents still consider as relevant the barriers dealing with security aspects hindering the inclusion of HiL in the CI/CD process, 31% do not consider such barriers as a real impediment. All three mitigations previously identified were considered relevant by the survey participants. Specifically, the adoption of test case prioritization techniques is predominant (31 out of 55 respondents), followed by the usage of simulators or mock-ups (28), and the usage of incremental builds (17).

Figure 4 shows the results of the survey in terms of the six challenges belonging to the Pipeline Properties category. Unsurprisingly, 47 out of 55 respondents consider the long build execution time as a relevant challenge. In addition, although from the semi-structured interviews the remaining five challenges were experienced by one or at most two different organizations, the survey indicates how some of such challenges are felt as relevant by more than 69% of our participants. These are (i) the need to properly estimate the build time before timing out the CI/CD process, (ii) the difficulty in properly configuring static code analysis tools, and (iii) the presence of a CI/CD configuration highly coupled with the environment. Regarding the impossibility of having access

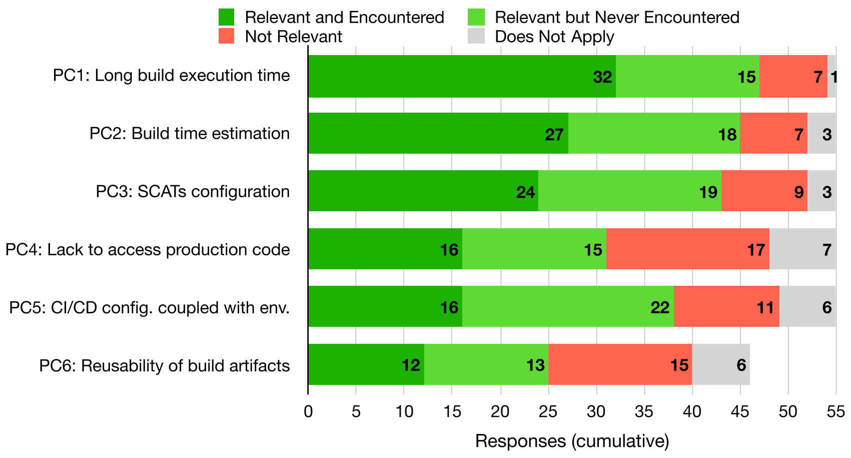

Fig. 4. Results of pipeline challenges perception: Pipeline Properties.

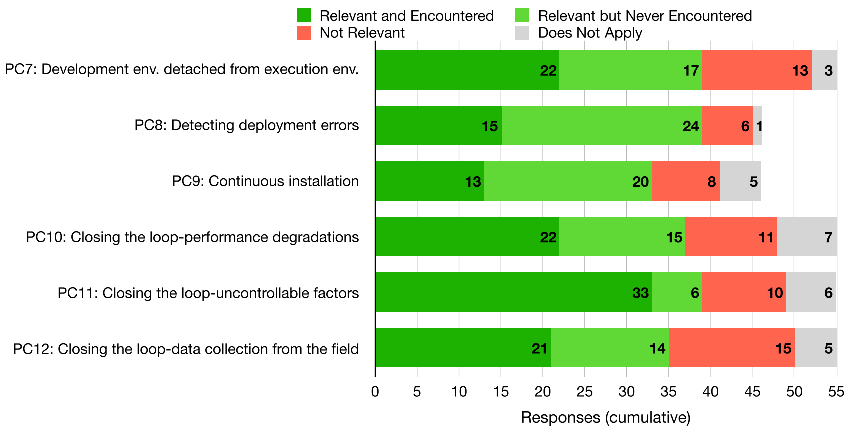

Fig. 5. Results of pipeline challenges perception: Thoroughness.

to the production code, if we do not consider the 7 participants reporting that this challenge cannot apply to their context, ≃31% of the respondents do not consider it as a relevant challenge.

Moving onto the mitigation strategies, more than half of our respondents (30) rely on test case prioritization techniques, 28 rely on parallelization, and 26 rely on nightly builds for time-intensive tasks, whereas 18 consider useful the adoption of incremental builds during normal working hours. Finally, 1 participant reported a new mitigation strategy dealing with long builds where *"we simulate faster than in the reality where possible"*; however, the same participant also points out the drawback of this mitigation—that is, having different build outcomes when using simulators and HiL: *"this can introduce subtle timing differences in the test results."*

For what concerns pipeline-related challenges in the Thoroughness category dealing with ensuring the overall accuracy and completeness of the CI/CD pipeline (see $PC_7$, $PC_8$, and $PC_9$ in Figure 5), differently from the $RQ_1$ results, more than half of our survey respondents consider the presence of a development environment detached from the execution environment as a real impediment to set up a CI/CD process for CPSs. This is also true for the difficulties in detecting deployment-related errors (39 respondents). The preceding differences stress the impossibility to have a "standardized" CI/CD configuration that can be applied to almost all CPS domains.

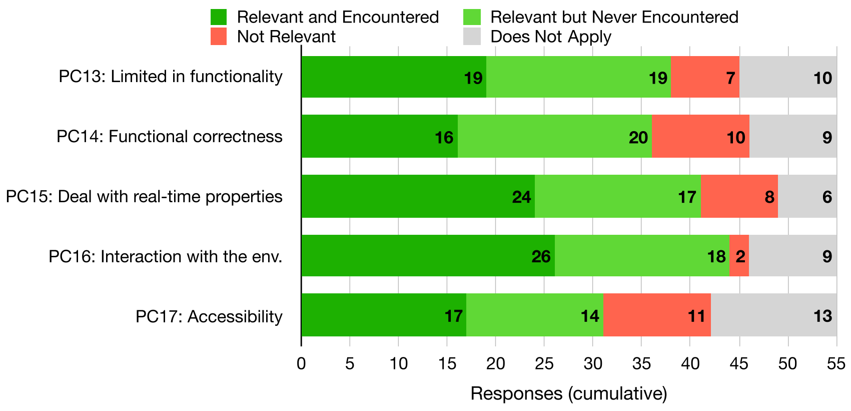

Fig. 6. Results of pipeline challenges perception: Simulators.

Although all participants considered the adoption of containerization a viable solution to overcome these challenges, one new mitigation strategy comes up from a survey participant, which, for $PC_8$, suggests the possibility of developing and adopting static analysis tools able to analyze (and detect errors from) deployment scripts.

Moving to the three challenges related to closing the DevOps loop by gathering data from the hardware (i.e., real environment), by looking at the bottom part of Figure 5, it is possible to state that more than 65% of the respondents consider them as relevant, with the presence of uncontrollable factors to account for having the highest percentage ($\simeq$ 71%). Although from $RQ_1$ we did not find any mitigation strategy for these challenges, we obtained some feedback from eight survey respondents. First of all, it could be possible to continuously analyze the logs also after the operation has started. At the same time, one survey respondent points out the possibility to make the monitoring less impactful on performance by *"disabl[ing] invasive logging methods."* For what concerns the presence of uncontrollable factors, one respondent pointed out how using continuous testing allows to *"better overcome the problem of the uncontrollable factors in real life systems and usually diminish the future costs and improve efficiency."* There are also mitigation strategies dealing with the overall CI/CD process. Specifically, one respondent mentions the possibility to use parallel DataOps observability pipeline: *"we use ELK, but it is still under debate/migration."* However, a different respondent highlights as possible mitigation the presence of *"cross-functional teams which bring in more collaborations and ideas."*

Figure 6 shows the results of the external validation survey for the challenges dealing with the inclusion of simulators in the CI/CD process. As can be seen from the figure, at least 36 out of 55 respondents considered such challenges relevant. The only exception is the challenge of dealing with the impossibility of accessing the third-party simulators adopted in the pipeline ($PC_{17}$). In this case, 13 respondents mention that it does not apply to their context, meaning that the simulators are mainly self-developed within the organization they belong to. In terms of mitigation, instead, (i) 28 respondents use a CI/CD process made up of both simulators and HiL for overcoming the presence of limited functionality, (ii) 24 use both simulators and HiL for overcoming the complexity due to a tight interaction among different components and the environment, and (iii) among the 31 respondents struggling with the simulators' accessibility, 14 adopt the "timeout" feature.

With regard to the four challenges dealing with the inclusion of HiL in the CI/CD process, as shown in Figure 7, the costs and scalability challenge is the predominant one (49 out of 55 respondents), followed by the need to check for HiL availability (41), and the complexity for automating both deployment and testing activities on HiL (42 and 40 respondents for $PC_{19}$ and

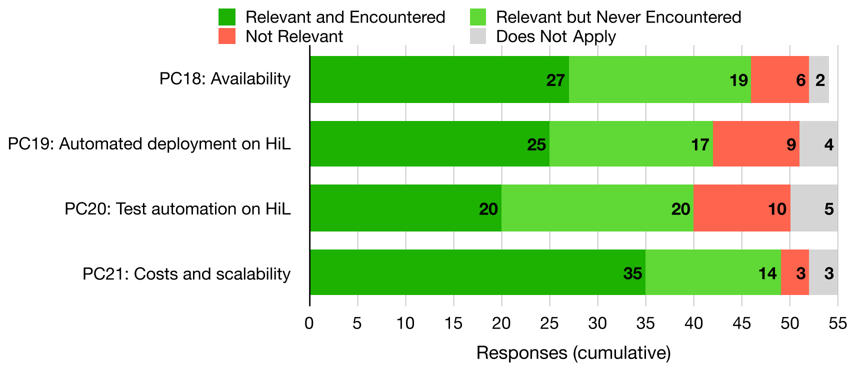

Fig. 7. Results of pipeline challenges perception: HiL.

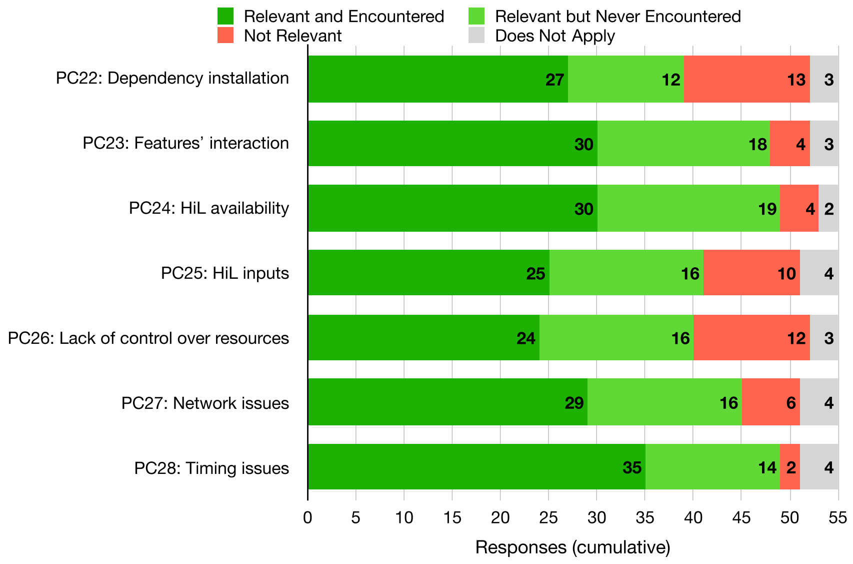

Fig. 8. Results of pipeline challenges perception: Flaky Behavior.

$PC_{20}$, respectively). No new mitigation strategy comes up from the survey results. However, 28 respondents confirm that the adoption of simulators and HiL in different build stages can help deal with costs and scalability issues, whereas 19 adopt the "green-build" rule—that is, HiL can only be considered when the CI/CD process relying on simulators has a green status.

The last category of pipeline-related challenges being validated through the external survey considers the root cause for flaky behavior experienced in the CI/CD process. As shown in Figure 8, for each challenge we have that more than half of our respondents consider it relevant for CPSs. Moreover, 49 out of 55 respondents consider it challenging to deal with HiL availability and simulators not coping with timing issues. Unsurprisingly, the two challenges being not specific to the CPS development (i.e., $PC_{22}$ and $PC_{26}$) are the ones where several respondents (13 and 12, respectively) mentioned that it is not relevant. Fixing the pipeline configuration is the most frequent mitigation strategy, as indicated by 31 respondents, but no further mitigation strategies are suggested.

Finally, by looking at the 15 answers to the open-ended question aimed at eliciting other challenges that we did not encounter in the semi-structured interviews, we gathered the following additional challenges:

(1) *Guaranteeing the supply chain security* (three respondents);
(2) *The impossibility to use simulated environments* unless the quality of specific data types is ensured (one respondent);
(3) *The need to have field tests included in the DevOps cycle even with a lower frequency* as both simulators and HiL can only cover a fraction of what usually happens during real field testing activities (one respondent);
(4) *The difficulty to implement a "quick-retry" feature* in the CI/CD process, to selectively rollback at specific stages mainly because this is highly dependent on the infrastructure language (one respondent); and
(5) *The difficulty to reduce the build execution time when dealing with HiL due to the need for checking the HiL availability*—that is, *"very long hardware boot times"* (one respondent).

## 5 DISCUSSION AND IMPLICATIONS

This section summarizes the main findings and implications of our study. We divide the section into implications for (i) developers, (ii) educators, and (iii) researchers.

### 5.1 Implications for Developers

We start by discussing what, based on insights learned from this study, developers must consider when trying to set up and evolve a CI/CD pipeline for CPS development.

*Simulators are necessary to achieve continuous builds on CI/CD pipelines.* Performing CI/CD on real hardware is often unfeasible, for different reasons. The automated deployment may be complicated, or the hardware may not be available on-site. In addition, organizations doing V&V tasks only may have limited/no access to hardware, simulators, or even to the production code. Therefore, simulation is often the only choice available. However, having a reliable simulator is challenging for many CPS developers. In some cases, simulators come from hardware producers; however, in other circumstances, the only option is to develop them in-house. This requires the allocation of suitable skills and efforts in the development process. Failing to do so would have severe consequences on the ability to set up not only CI/CD but also even simple test automation without relying on the hardware directly, when this is possible.

*Balancing the use of simulators and HiL in the pipeline.* Deploying and running CPSs on HiL at every change could be troublesome and expensive, and may result in slow feedback. At the same time, for the reasons mentioned before, it is unlikely that developers could fully trust a quality assessment performed solely on simulators. Therefore, it is highly desirable to configure staged builds relying on different execution environments, namely (i) continuous builds on simulators, aimed at providing fast feedback to developers (e.g., about the outcome of static checks, or possible integration issues discovered by tests), and (ii) periodic (e.g., nightly) builds on HiL, to verify whether the assumptions made on simulators are still valid, checking properties that often cannot be verified on simulators (e.g., response time properties), testing the system in scenarios that cannot be easily simulated, or verifying the compatibility of the software against hardware variants not fully reproduced by the simulators.

*Late delivery is the crux of CPS development.* For the reasons explained previously, CPS software tends to reach target production hardware very late in the development. This has several negative side effects, including the late discovery of defects that could not be identified through simulation, but also having a system that reaches the end user very late. Allocating sufficient effort, resources, and competences to enable automated delivery is therefore highly desirable.

*Having hardware experts on-board may be a plus.* Based on what we learned from this study, it is clear how CPS development may highly benefit from the availability of both software and hardware experts so that it is much easier to self-develop simulators whose behavior is as much as possible like the one of the real device. This would help reduce the differences that might be observed in terms of build outcome, such as the number and type of failing tests, when running the process on simulators and HiL. However, the presence of hardware experts in a team, when available, has been found to be useful by our interviewees. Yet a context in which both hardware and software evolve makes tasks such as change impact analysis more challenging to handle—for example, to determine whether and to what extent a hardware change would impact some software components, or some software evolution would hinder the integration of certain pieces of hardware.

## 5.2 Implications for Educators

In the following, we discuss what, based on insights learned from this study, would be expected for what concerns the creation (or enhancement) of curricula related to CPS development.

*Blended curricula with hardware and software competencies.* Any effective DevOps organizational setting or management of a CI/CD pipeline likely requires software engineering expertise other than what is currently taught in regular graduate-level courses, such as knowledge about the hardware, software-hardware interplay, and domain standards expertise. On the one hand, university curricula shall strive to include such aspects in their teaching. On the other hand, practitioners should more actively engage in standardizing CPS application lifecycle management practices, patterns, and tools to enable the aforementioned educational augmentation exercise.

*Specialized courses on simulator development.* In a context for which CPS specific curricula are highly desirable, one competence assumes paramount importance, and this is the development of simulators. The latter requires combining knowledge from physics, automated control (e.g., system dynamics, discrete systems), and virtual reality (many simulators leverage 3D or even virtual reality environments, similar to those used in video games).

*Teaching CI/CD in complex, heterogeneous environments.* CI/CD is oftentimes taught in the context of conventional system development. To favor the adoption of CI/CD for complex systems, particularly for CPSs, courses on CI/CD should touch on topics related to (i) coping with complex hardware or simulators attached to the pipeline, and (ii) pondering fast builds with the need for testing a CPS on multiple devices (or simulators), where this is appropriate. In addition, although conventional CI/CD literature advocates "building at every change" [13], CPS developers need to face reality, and therefore such a common wisdom need to be revisited. Similarly, we found that for large and complex CPSs, "retest all" does not work, and therefore incremental builds are a widely adopted practice.

*Software architectures for CPSs.* CPSs heavily interact with HiL interfaces (and sometimes multiple HiL, having different characteristics and varying APIs) and, during the development process, with simulators. The latter may be updated or even replaced by better ones. From an educational perspective, it is desirable that courses related to software architectures properly treat such scenarios, discussing the proper architectural choices or design choices allowing an easy (even at runtime) replacement of different kinds of HiL and simulators in the software systems. Software components of CPSs may need to be deployed on, or interact with, multiple types of devices (e.g., a control software may be deployed on different car models). This requires that developers must have suitable knowledge of product line engineering and follow related practices when designing CPSs. Furthermore, it is desirable to teach prospective CPS developers about how to design a CPS architecture to make a system scalable, but also secure, and easy to be monitored and tested.

## 5.3 Implications for Researchers

Implications for researchers aim at developing approaches and tools to support developers in setting up, maintaining, and using CI/CD pipelines for CPSs.

*The target environment of CPS is multifaceted and diversified, making CI/CD pipelines complex and expensive.* Often a CPS may target multiple devices, as well as both HiL and simulators. This may entail a build matrix against which the pipeline must be run—that is, the matrix describes different combinations of parameters (e.g., simulator models, HiL instances, other settings) for the build. That being said, it is possible that although some changes may entail different behavior on different matrix instances, other changes do not, and therefore running the build on all possible configurations would be a waste of resources. On the one hand, this stimulates research toward approaches aimed at recommending the creation of a suitable build matrix system based on similar systems, and in general systems targeting similar devices. In addition, these kinds of recommenders should be able to point out the need for maintaining build matrices by learning "from the crowd" (e.g., the need to prune out obsolete environments and add new ones). On the other hand, proper approaches should be developed to trigger builds on different matrix instances based on the changes performed.

*Coping with multiple root causes for flaky behavior.* The complex technological stack, the behavior of simulators and HiL, their (sometimes uncontrollable) unavailability or lack of accessibility, and the mechanisms used to collect test outputs (e.g., sensors or video cameras) require not only to better monitor all possible elements causing flakiness but also to combine and enhance various mitigation approaches, including checking the status of HiL/simulators, and leveraging the "usual" retries. As indicated by the participants, flaky behaviors in CPSs are often due to the complex interacting environment (e.g., lack of complete control on the hardware status) rather than on the order with which the tests are executed. Hence, flaky test detectors that target flaky tests considering their ordering [24, 88] are not effective for environment-dependent flakiness. CPS-specific detectors could be inspired to those for undetermined specifications [87], or based on ML models [59] but trained on CPSs data and encompassing CPS-specific features, such as changes to simulators or HiL configurations, as well as their build logs. To improve CI/CD infrastructures for CPS, it may be useful to develop recommenders, integrated into the pipeline, that are able to support developers in the identification of flakiness behavior and identify its root causes.

*Challenges in automated test execution.* We found that one of the reasons that impede full CI/CD automation for CPS is the difficulty to automate test execution, especially when the system is deployed on the hardware. In other words, the system receives inputs from sensors and interacts with actuators. Full test automation requires (i) tools, such as scenario generators or record replay tools able to seed inputs to the CPS, and (ii) the capability of CI/CD infrastructure to support the execution of such tools. Very often, these tools are GUI oriented and not particularly well suited to be integrated in a CI/CD pipeline.

*Challenges in automated oracle creation.* The CPS execution environment (e.g., simulators or HiL) drastically complicates the definition and automatic check of oracles. The latter requires to ponder several factors: (i) the test scenario (or requirement to assess), (ii) the accepted level of realism in simulations, (iii) the readiness level or maturity of the hardware proxies used in the pipeline, and (iv) the output sources (e.g., based on actual sensors' data or mocking/synthetic data). Besides, the oracles consist of value ranges (e.g., time intervals) instead of scalars, or they may be signals that need to be properly processed, as highlighted in existing studies on testing for CPS [6, 53]. It may be important to account for non-functional properties, including timing ones [80]. In addition, to cope with inputs originating from sensors or even from a multimedia recording of the CPS execution (as pointed out by $O_3$ and $O_4$), it is desirable to develop approaches for pattern recognition [28, 38, 72].

*Need for specific fault models.* Looking more broadly at configuring V&V phases within the CPS pipeline, respondents would like to discover early some defects through static analysis. This requires a clear fault modeling in the CPS context (as the ones for autonomous cars [26] and unmanned vehicles [81]), but also to develop CPS-specific linters, which can be integrated into the CI/CD pipelines to allow early detection of build failures, hence avoid to perform expensive testing activities, and hence long builds, which constitute a major problem for CPS developers according to our study results. Moreover, CPS-specific fault models can be useful for other purposes, not only to facilitate root-cause analysis [26] but also to create domain-specific mutation testing strategies, as has been done in other cases such as deep learning [36] or mobile development [19, 76].

## 6 THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observations. The interview participants might have misinterpreted our questions, or they might have reported their personal (and biased) views of the phenomenon. Although this is typical for interview-based studies [17, 33], we mitigated the threat by using semi-structured interviews and following up with clarifications every time we realized this was needed.

There could be threats to construct validity related to how survey respondents interpreted the survey questions and provided their answers. We mitigated this threat by providing a self-explanatory description of the challenges, barriers, and mitigation strategies. In addition, we left them the possibility to provide open comments to also point out cases of misunderstanding. However, based on the provided answers, we had no evidence of cases where respondents had difficulties in understanding the posed questions. As well, for the external survey, we leveraged demographic information to filter out responses where the information provided made it evident that a participant did not have the required knowledge.

Threats to *internal validity* concern confounding factors that could have influenced our results. To limit subjectivity in our coding, we employed multiple coders, computed inter-rater reliability, and used follow-up discussions not only to resolve cases of inconsistent coding but also to review any single coding. We elicited codes and relations only based on explicit occurrences of words in the transcripts. However, we could not exclude imprecision due to our interpretation of the participants' answers.

Another threat could be the low representativeness of the respondents in the semi-structured interviews and, to some extent, in the external survey. In the first case, participants were obtained through personal contacts, as we need people available to participate in a relatively long interview. However, such participants cover a relatively diversified set of domains (8). As for the survey, the use of snowballing and especially the use of *Prolific* allowed us to mitigate a possible bias due to the direct personal contacts.

Threats to *reliability validity* relate to the extent to which results can be reproduced. To achieve this goal, we (i) describe the data collection and analysis process in detail and (ii) provide in our replication package the detailed outcome of the coding phases.

Finally, threats to *external validity* concern the generalizability of our findings. The interview-based study has been conducted involving 10 organizations developing CPS for eight different domains. We are aware that the obtained findings may not generalize to different organizations and domains. Indeed, from the performed interviews, we found that CI/CD pipelines were extremely different from case to case. Therefore, as in other interview studies conducted within a limited set of organizations, and also considering the study topic, the generalizability is relatively limited. To mitigate this threat, when addressing $RQ_2$, we validated the findings collected in $RQ_1$ through an external survey with practitioners different from the ones involved in our semi-structured interviews, and belonging to nine domains. Still, it is possible that, also in this case, as Figure 2 shows, some domains are better covered than others, and some are still not covered at all.

## 7 CONCLUSION AND FUTURE WORK

In this article, we investigated the adoption, usage, and evolution of CI/CD pipelines for CPS development by focusing on challenges and barriers that DevOps teams face when setting up or evolving CI/CD processes for CPS development, highlighting that the configuration is highly dependent on the domain. The study is based on interviews from 10 organizations developing CPSs in eight different domains, followed by a member-checking survey within the same development teams, and an external validation survey involving 55 participants from nine domains. By performing an open coding on the interview results, we elicited a set of challenges/barriers, along with their mitigation strategies.

The obtained findings are a first step toward supporting DevOps teams in properly using and configuring CI/CD for CPSs. In addition, they have implications on how to enhance education/training for CPS developers and trigger future research. Based on that, future work aims at triangulating this study through other channels, such as in-field observations, and at investigating bad practices in applying and maintaining CI/CD for CPSs. In particular, our goal will be to automatically detect, by analyzing CI/CD pipeline configurations and runtime data, problematic situations (“smells”) that would require an intervention on the DevOps side, and, for what is possible, automatically suggest repairs.

## APPENDIX

## A DETAILS ABOUT THE INTERVIEW PARTICIPANTS

The interviews’ transcripts together with the labeling procedure helped us in forming organization profiles, focusing more on how the interviewed companies set and maintain a pipeline for CPS development.

In the following, we detail the development process of the interviewed organizations, focusing more on the status of their CI/CD pipeline in terms of (i) build triggering strategies (e.g., continuous or periodic), (ii) co-existence of multiple pipeline configurations (e.g., for different devices) and the frequency of changes occurring to them, (iii) the phases being automated/executed within the pipeline, and (iv) the usage and setting of HiL and simulators within the pipeline (e.g., a pipeline can be a mix of different environments used in different circumstances). In the following, for each organization participating in the interviews, we describe—by leveraging the codes elicited during the interviews’ transcripts analysis phase—the CPS development process and especially the adoption of CI/CD pipelines and, in general, of build automation.[3]

### $O_1$ (Aerospace)

Context: $O_1$ is involved in V&V tasks for aerospace software (i.e., on-board software for satellites), hence their CI/CD pipeline is only for V&V and not for development. This is because, due to the safety integrity level of the CPS, the development and the V&V teams and pipelines must be kept distinct [22]. $O_1$ relies on conventional programming languages dictated by standards in the aerospace domain (*“We mainly use ANSI C-99 following the MISRA rules”*). This implies the need

[3]The interested reader can find the code mind maps in the replication package [84].

for certifying software (i.e., following the MISRA (Motor Industry Software Reliability Association) standards [1, 7]).

Pipeline Status: $O_1$ started adopting CI/CD practices less than 1 year ago, mainly due to a limited culture within the team about CI/CD principles. Moreover, it does not have a strict separation of roles for what concerns the type of interaction with the pipeline (*"a developer who needs to customize a CI/CD pipeline by simply using yaml files can customize it directly"*). $O_1$ does not rely on build matrices with jobs related to different environment variants since *"the pipeline does not have to change/evolve based on the changes in the technologies being used (version for compilers and or programming languages), the aerospace domain follows the waterfall process. So everything is [frozen]: no changes may occur later on in the process."*

Automated Tasks: Due to the application domain and the related standards and certification constraints, the pipeline compiles the software provided and developed by the customer, relies on SonarQube for (i) checking the fulfillment of the MISRA rules for certification, (ii) identifying maintainability problems (i.e., *"we also have non-functional requirements expressed in terms of rules available in SonarQube"*) mainly related to the presence of duplicated code, and (iii) identifying bugs as soon as they are introduced, and executing unit and robustness tests to *"check how the system behaves/reacts in the presence of unexpected inputs ([e.g., ] inputs having values out of the admissible range)."* Furthermore, considering the overall scope of the pipeline, its triggering is manual, even if there are also nightly builds used for running test suites requiring a long time to complete. It is important to note that the testing criteria to derive the test cases to include within the pipeline are expressed from the customer as non-functional requirements (*"i.e., use MC/DC for deriving the test suite"*). Finally, $O_1$ has to consider time constraints for the pipeline setting to deal with possible issues that may arise when launching the simulator (e.g., memory leaks or impossibility to access the simulator).

HiL and Simulators: $O_1$ cannot involve HiL in the pipeline, as it would require a clean room not accessible from the outside. Instead, it relies on third-party simulators dictated by the customer. This is because the customer follows *"a framework for simulation aimed at hosting different simulators for different satellite models for the digital twin of the satellite."* Relying on third-party simulators helps in reducing the costs/efforts needed to develop the simulators from scratch, as well as helps in guaranteeing the trustworthiness of the outcome being produced and provided to the customer. Of course, the level of trustworthiness increases for those cases where the simulator is provided by the same vendor of the hardware device that must be simulated.

### $O_2$ (Healthcare)

Context: $O_2$ is a large organization involved in the healthcare domain: it provides CT scanners for clinical use. With regard to the development process, $O_2$ has a team for each component being developed—around 17 different teams working on 70 branches—together with an integration branch where all of the other branches are integrated into a *"single joined point."* Furthermore, each team adopts conventional programming languages (i.e., mainly C# and C++).

Pipeline Status: $O_2$ already has a CI/CD pipeline in place for CPS development that was introduced 4 years ago, and they are still improving it. Furthermore, based on its application domain, $O_2$ is constrained to *"follow medical application frameworks providing a base set of rules in terms of how to build applications and how to integrate them."* The latter requires the adoption of processes aimed at verifying whether or not the overall development process adheres to the regulatory standards for developing medical applications.

Automated Tasks: $O_2$ adopts both incremental and nightly builds. Of course, the tasks involved in the different types of builds, as well as the execution environment involved in them, vary. Specifically, nightly builds leverage HiL and run three different types of testing, namely unit/component, sub-system, and system testing. To provide developers fast feedback about the impact of their changes, $O_2$ relies on incremental builds executing only a subset of the whole set of functional tests—by doing *"impact based testing to figure out the impact of the changes and select the tests to be executed based on the impact."* To control the overall build execution time, $O_2$ encourages developers to push small changes leading to *"small sets of tests to be executed."* Finally, both incremental and nightly builds run static code analysis tools mainly aimed at identifying maintainability and security flows in the code.

There is a specific type of build aimed at checking performance requirements like *"test whether each component (some components) stays within the resource limits they are assigned to."* The outcome of such a build is compared over time to identify and monitor possible performance degradation within the whole system. Moreover, $O_2$ has a specific DevOps team for checking the fulfillment of security requirements, even if this is not done continuously while only *"near the finalization of the product,"* and it is not automated.

HiL and Simulators: As explained earlier, both the triggering strategies adopted by $O_2$ and the tasks being automatized within each type of build influence the choice between using simulators and/or HiL. Nightly builds have an automated deployment on a "real" CT scanner *"without reusing existing artifacts while building all of them from scratch in a clean environment,"* for executing the whole test suite in a real production environment. Note that when talking about "real" CT scanner, $O_2$ refers to *"physical systems that are equivalent to the real hardware in the CT scanner but not connected to anything around it which has a simulator running on it."*

For what concerns simulators, $O_2$ relies on self-developed simulators —there are suitable knowledge and skills to properly develop simulators (i.e., $O_2$ develops both the software and the hardware). However, at the moment, $O_2$ does not use simulators (*"mainly used for functional testing only"*) for checking non-functional (i.e., performance) requirements.

## $O_3$ (Acoustic Sensors)

Context: $O_3$ is involved in CPS innovation for the industry, among others the development of the SPL Noise Meter Board—a low-cost, high quality, electronic sensing board capable of measuring noise of the environment. It does not have any separation of roles between the members of the team (*"The team is the company"*); however, the team is composed of both software and hardware experts who work together, simplifying the overall development process, particularly for those activities requiring the integration and communication between software and hardware components (*"Useful for sensors' integration . . . [it] help[s] having knowledge about the hardware components, how they work and how it is possible to communicate with them"*).

$O_3$ adopts a **Pull-Request (PR)** development process with one branch per feature (i.e., *"Several branches for maintaining and developing different features"*). Furthermore, even if it does not have strict guidelines in terms of coding standards, $O_3$ attempts to adopt similar coding styles within each branch. Finally, it relies on conventional programming languages, such as Python for testing and C and C++ for micro-controllers development.

Pipeline Status: At the moment, $O_3$ does not have a CI/CD pipeline for CPS development.

Automated Tasks: Even if $O_3$ does not have a CI/CD pipeline in place, the deployment is fully automated, although the testing is still manual mainly due to the impossibility of automating the oracle specification, particularly for testing acoustic signals. Furthermore, even if $O_3$ does not have certification constraints for the developed code, they need to cope with certification constraints *"for the acoustic signals."*

HiL and Simulators: $O_3$ only uses real hardware devices, even if within the organization there is the wish of including simulators in the process to test the acoustic signal (i.e., the main outcome of their product) in a controlled environment (i.e., *"removing noise from the surrounded environment"*).

**$O_4$ (Robotics)**

Context: $O_4$ is involved in the development of autonomous robots, and is made up of several development teams where each team accounts for both hardware and software developers. Furthermore, it adopts a PR development process with one branch per feature (i.e., *"We have a branch for each feature that needs to be implemented and/or improved and we use PRs to merge the work in the stable release branch"*). Based on the application domain, it mainly adopts C++, together with Python for users' interfaces and for interacting with the hardware devices.

Pipeline Status: $O_4$ has a fully containerized (using Docker) pipeline for CPS development. It relies on continuous and nightly builds, even if they are not used for running time-intensive tasks (i.e., *"that is not so expensive in terms of execution time"*), while for running regression testing activities on already packaged components and for deployment to the customers. Furthermore, the CI/CD configuration is pretty stable, meaning that even if each branch may rely on a customized CI/CD process, the configuration does not have to change over time.

Automated Tasks: Our interviewee mentions the execution of static code analysis tools to inform developers about code quality degradation and unit tests relying on simulators. Only when a PR is peer reviewed and there are no failures in the entailed CI/CD process is it possible to merge the change on the stable repository and enact a release process for shipping the product to the customers. $O_4$ monitors the overall quality of the development process in terms of static analysis metrics and code coverage from the unit test execution. The application domain does not introduce certification constraints (i.e., *"If you want to sell a robot you do not need to have a certified robot"*), but it hinders the automation of non-functional testing within the pipeline. Specifically, our interviewee mentions the manual execution of reliability and safety tests *"running the robot a long time with a guy supervising the test execution to understand when and why the robot starts to not work anymore"*, or *"guaranteeing that once pressing the stop button the robot actually shuts down."*

HiL and Simulators: $O_4$ relies on third-party simulators and HiL. One point raised by our interviewee is related to the partial usage of Docker on the hardware so that it is possible to run the robot in a privileged mode and switch between software versions quite easily: *"each one may choose the version of the software that has to be run over the robot."*

**$O_5$ (Automotive)**

Context: $O_5$ is a large company operating in the automotive domain working on the software-focused driving platform. The development process is organized into three different teams each one with a specific goal: *"one working on virtual machines, one working on web services because we provide DevOps solutions for embedded systems, and finally, we have a small team working on customer delivery with the goal of adapting our tools to the customers' needs."* This is the only organization in our study relying on real-time languages (i.e., real-time Java) to cope with scheduling requirements of embedded systems.

Pipeline Status: $O_5$ already has a CI/CD pipeline in place mainly for deployment purposes (*"we are able to support updating software on devices on the fly"*), even if it is working on improving it—*"it is still kind of an infancy we are still working on improving."*

Automated Tasks: $O_5$ mainly uses the pipeline for deployment. However, differently from other organizations, $O_5$ relies on virtual machines instead of using containers for several reasons: (i) better control over the resources (i.e., *"the ability to enforce our resource usage inside the virtual machine while you do not have quite the same extent with a container"*), (ii) versioning capability (i.e., *"when a new service comes in, you register it so it is easy to start a new version of this service, run down the old one and switch over the new one during run-time"*), and (iii) memory safety guarantee, (i.e., *"by looking at a recent post by both Google and Microsoft we found that around 70% of the security violations are due to failures of memory safety. So by using a garbage-collected environment, we can prevent those issues from occurring"*). Going deeper into how the deployment process works, $O_5$ first creates the virtual machine (i.e., emulating the virtual environment), then the OSGi infrastructure, and finally it tests individual modules. The latter means that $O_5$ does not test all the developed modules together since it *"deploy[s] individual bundles to a platform."*

For what concerns the verification of non-functional requirements, $O_5$ performs security and performance testing, even if they are not included in the pipeline. Specifically, for real-time systems, it is important to monitor the impact of each change on performance properties to be able to identify, as soon as possible, the change introducing performance degradation (i.e., *"we have various performance tests that we run regularly to track our performance as the system evolves"*).

HiL and Simulators: Since $O_5$ develops software for embedded entertainment in the automotive domain, HiL is only available for a final validation on the customer's side: *"then employ our customer for the last mile,"* so most of the work is done relying on virtual environments.

### $O_6$ (Aerospace)

Context: Similar to $O_1$, $O_6$ operates in the aerospace domain, and is mainly involved in the development and refining of the routing algorithm for the FRA. For what concerns the programming language being adopted, $O_6$ relies on conventional languages: *"C and C++ [are] used for the back-end."*

Pipeline Status: $O_6$ already has a CI/CD pipeline mainly for deployment and testing purposes, which is under continuous improvements. Moreover, our interviewee mentions that the pipeline is more an MLOps than a simple DevOps pipeline.

Automated Tasks: Among the phases being automated, there are (i) static code analysis for identifying maintainability flows and spotting bugs as soon as they are introduced, (ii) unit testing, (iii) integration testing, and (iv) deployment. Furthermore, the execution of non-functional testing activities is mainly carried out manually and outside the pipeline, due to the high complexity of the real-time operating system under development. Similarly to $O_1$, it is required that the developed code satisfies strict certification requirements that are mainly checked by relying on code coverage tools. Differently from other organizations, $O_6$ does not rely on nightly builds, meaning that also time-intensive tasks are executed at each change (i.e., *"even the slow builds are continuously built"*). $O_6$ recommends that developers use private builds before pushing their changes on the stable release branch, at least for what concerns the execution of unit testing. Finally, the pipeline provides a monitoring mechanism for what concerns aspects of the real-time operating system such as scheduling and memory that *"gives us the possibility to collect feedback/evidence that may help us in obtaining the certifications."*

HiL and Simulators: $O_6$ relies on both simulators and HiL; however it does *"not have simulators and HiL in the same pipeline mostly for certification issues."* Specifically, it is possible to rely on real devices only when there is enough trustworthiness about the software in terms of correct behavior, as well as the absence of crashes gained by relying on self-developed simulators.

### $O_7$ (Railways)

Context: $O_7$ is involved in delivering software for railways (i.e., TCMS), and similarly to what is reported for the aerospace domain, due to the safety integrity level of the software under development, developers and testers must be different (i.e., *"Testers and Developers are in separate teams in presence of new functionality to be implemented both start together to implement and write test cases"*).

Pipeline Atatus: $O_7$ already has a CI/CD pipeline in place for CPS development—*"introduced two years ago"*—that, at the moment, is in a continuous improvement state since it does not automatize the whole development process (i.e., *"The deployment on the real train or on the hardware test track is not automated at the moment even if we are working on making it automatic"*). In terms of programming language, the interviewee mentions the need of adapting the programming language to the device on which the software has to be executed; however, they mainly rely on conventional languages.

Based on the application domain, $O_7$ adopts staged builds following the "green-build rule." In the first stage, the build process is executed on a virtual machine (i.e., *" [a] virtual train, software running on a PC that should behave like it does on a real train"*). Once a change occurs on a specific component, the related build process is enacted and, in the presence of a green status, all the components are deployed together so that it is possible to enable the execution using the virtual train (*"the devices are run in some kind of containers and we have frameworks building and connecting the whole set of devices and components"*). If the build process ends with a successful state, it is possible to move to the next stage that relies on the hardware test track (i.e., *"where we have the whole set of devices and even some more that we do not have in the virtual train"*). Finally, if the build process for the second stage ends with a green status, it is possible to run the last stage relying on a real train. Note that each device/component has a proper CI/CD configuration.

Automated Tasks: $O_7$ uses the pipeline to automatically test basic functionality (i.e., *"We test specific train functionality such as whether we should activate the train in [a specific] mode"*), as well as the interaction between different components/devices (i.e., *"we have a long sequence of events for each test that involves different devices and components so we are mainly doing integration testing"*). The test suites used in different stages of the build process may be different since *"for some test cases, we are not allowed to rely on the virtual environment while we must consider the hardware track or a real train."* At the moment, $O_7$ has automated deployment within the pipeline only for the first stage—that is, relying on the virtual train (for which the overall build execution is *"around one hour and [a] half"*)—whereas it is done manually for what concerns the other two stages: hardware test track and a real train. Testing against non-functional requirements is also done manually, because of the high variability and complexity of the environment.

Going deeper into how developers interact with the CI/CD pipeline, $O_7$ enforces developers to run private builds before pushing their changes on the main stable repository. The private builds are aimed at executing the same test suite later executed on the CI/CD servers: *"for the moment we cannot configure the number and type of tests to be executed locally."* Furthermore, the "green-build rule" is used for determining the development tasks: *"in presence of a failure all developers are stopped until the build becomes green again."*

HiL and Simulators: $O_7$ adopts both simulators and HiL in different stages of the build process, with the use of HiL occurring only in the last stage of the pipeline.

### $O_8$ (Railways)

Context: The application domain of $O_8$ is railways, and particularly the development of a specific component used for transmitting data between on-board and ground applications. $O_8$ uses C and C++ (i.e., conventional programming languages), and it has strict constraints for what concerns the production code that has to satisfy strict certification requirements, as well as the compliance with the railway standards and specifications. The latter is mainly checked by relying on *"a specific complex tool that can be configured based on specifications and standards."*

Pipeline Status: Due to the limited availability of human resources together with the complexity and the safety integrity level of the application domain, $O_8$ does not have a CI/CD pipeline in place for CPS development, and it has, in general, little automation in the development process.

Automated Tasks: Only the adherence to standards and specifications is automated, whereas functional *"tests are written manually starting from requirements and system specification[s] but also their execution requires a manual effort."* This is also the case for non-functional and integration testing (i.e., *"we have a set of testers in front of a screen who monitor and check for the presence of any discrepancies about what is expected and what is instead observed while running the system"*). Due to the effort and time needed to manually verify the reliability of the software under test, functional tests are executed at every change, whereas integration tests are only executed when the change impacts the *"interfaces with other modules/components."*

HiL and Simulators: Due to the high cost of the hardware devices involved in this particular application domain, $O_8$ mainly relies on simulators that are self-developed (*"we do not rely on third party very expensive simulators"*). However, once per week, $O_8$ performs a testing session with a real *"train running in a real environment with real traffic [and] possibly without people."*

## $O_9$ (Identification Technology)

Context: $O_9$ is involved in *"develop[ing] software relying on identification technologies such as RFID [Radio Frequency IDentification], Bluetooth low energy or bar codes,"* other than mobile app development for which there is a CI/CD pipeline used for testing and automated deployment on the play stores. For what concerns the CPS development, $O_9$ relies on conventional programming languages such as C# and Java.

Pipeline Status: The limited availability of human resources, together with a lack of culture for setting a pipeline dealing with sensors and actuators, results in not having a CI/CD pipeline in place for CPS development.

Automated Tasks: The testing phases are almost fully automated. Specifically, there are *"RFID-readers connected to a network"* on which it is possible to execute unit and integration testing activities automatically. For what concerns integration testing, it is important to remark that there are cases requiring the manual intervention of the tester (i.e., *"For instance, when we need to test a transfer of tags between different antennas we cannot use automation"*), as well as cases where it is required to interact with the hardware devices that cannot be simulated. Of course, in this specific setting, it is not possible to guarantee the overall reproducibility of the results of the test; however, *"the reproducibility of the test in this context is not required."* Furthermore, $O_9$ does not run the whole test suite at each change. Instead, they manually select some test cases based on impact analysis: *"select what are the test cases that are impacted by the change that, consequently, need to be executed."* Other than having unit and integration testing activities, $O_9$ also executes, from time to time, performance testing.

For what concerns the deployment of CPS-related software, $O_9$ relies on Docker for creating images that are manually deployed onto the servers.

Finally, the development process also features a monitoring component for the internal development platform and customers' devices, to notify about anomalies and errors, as soon as they occur.

HiL and Simulators: The development process adopted by $O_9$ relies on both (self-developed) simulators and HiL. Simulators are developed based on specific organization needs and use case scenarios, implying that they are limited in their functionality.

## $O_{10}$ (Energy)

Context: $O_{10}$ is involved in the development of prototypes and proof of concepts for the energy domain. The development of prototypes rather than real products represents concrete facilitation, since there may be less stringent constraints in terms of pipeline setting and evolution.

Pipeline Status: What is mentioned previously justifies the presence of a mature (i.e., introduced in 2016) pipeline adopted within the organization for CPS development that uses conventional programming languages, mostly Java and Python. The pipeline configuration is pretty stable probably due to the development of prototyping solutions that do not need to be shipped to real environments.

Automated Tasks: Other than having a compilation phase, the CI/CD pipeline is aimed at executing unit and integration tests (*"Our pipeline is mostly for unit testing (80%) but there is also some integration testing"*), followed by a deployment phase where the packaged version of the software is usually stored into an artifact repository as a Docker image. Furthermore, safety requirements, such as checking that a battery is not charged more than a certain rate, are specified and checked through unit test cases that do not involve the real devices. Thanks to the need for developing prototyping solutions, the pipeline accounts for static code analysis tools and linters that are mainly used for checking maintainability issues only (i.e., *"They are not used for checking out bugs, but mostly for making sure that the code is easy to read for other colleagues and for maintainability purposes"*). Moving the attention on the triggering strategies, $O_{10}$ does not rely on nightly builds. It only uses incremental builds so that each build execution time does not overcome the 10-minute rule.

HiL and Simulators: Looking at the execution environment, $O_{10}$ does not need to run the software on embedded devices, meaning that it *"tr[ies] to find devices having interfaces to communicate with. So basically we run our software on a traditional machine and it just communicates with the hardware."* So, differently from other organizations, $O_{10}$, other than simulating the hardware when needed (i.e., *"we simulate the battery for testing the charging protocol"*), mainly replaces it with mock-ups (i.e., *"it is very easy to mock a client just to see if our software sends the right commands or does not use any register twice"*). Only when the real devices are available and it is safe to use them for testing does $O_{10}$ use Docker images for checking the correct behavior over the real devices as well.

## ACKNOWLEDGMENTS

We would like to thank the study participants for taking their time to participate in the study and for providing insightful feedback.

## REFERENCES

[1] MISRA. 2004. Guidelines for the use of the C language in critical systems. Retrieved December 10, 2022 from https://www.academia.edu/26806881/MISRA_C_2004_Guidelines_for_the_use_of_the_C_language_in_critical_systems.pdf.

[2] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. 2015. A survey on testing for cyber physical system. In *Testing Software and Systems.* Lecture Notes in Computer Science, Vol. 9447. Springer, 194–207.

[3] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2021. Which commits can be CI skipped? *IEEE Transactions on Software Engineering* 47, 3 (2021), 448–463.

[4] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'18).* IEEE, Los Alamitos, CA, 143–154.

[5] Zuleyha Akgun, Murat Yilmaz, and Paul M. Clarke. 2020. Assessing application lifecycle management (ALM) potentials from an industrial perspective. In *Proceedings of the 27th European Conference on Systems, Software, and Services Process Improvement.* 326–338. https://doi.org/10.1007/978-3-030-56441-4_24

[6] Aitor Arrieta, Shuai Wang, Ainhoa Arruabarrena, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2018. Multi-objective black-box test case selection for cost-effectively testing simulation models. In *Proceedings of the Genetic and Evolutionary Computation Conference.* 1411–1418.

[7] Roberto Bagnara, Abramo Bagnara, and Patricia M. Hill. 2018. The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software. In *Proceedings of the International Static Analysis Symposium.* 5–23.

[8] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering* 27, 4 (2022), 1–31.

[9] L. Chen. 2015. Continuous delivery: Huge benefits, but challenges too. *IEEE Software* 32, 2 (2015), 50–54.

[10] Lianping Chen. 2017. Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software* 128 (2017), 72–86.

[11] Yingnong Dang, Qingwei Lin, and Peng Huang. 2019. AIOps: Real-world challenges and research innovations. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE'19)*. 4–5. https://doi.org/10.1109/ICSE-Companion.2019.00023

[12] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning (CoRL'17)*. Proceedings of Machine Learning Research, Vol. 78. PMLR, 1–16.

[13] Paul Duvall, Stephen M. Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley.

[14] Paul M. Duvall. 2010. Continuous integration patterns and antipatterns. *DZone Refcard #84*. Retrieved November 26, 2022 from http://bit.ly/l8rfVS.

[15] Paul M. Duvall. 2011. Continuous delivery patterns and antipatterns. *DZone Refcard #145*. Retrieved November 26, 2022 from https://dzone.com/refcardz/continuous-delivery-patterns.

[16] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'19)*. 830–840.

[17] Omar Elazhary, Margaret-Anne D. Storey, Neil A. Ernst, and Elise Paradis. 2021. ADEPT: A socio-technical theory of continuous integration. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSE (NIER)'21)*. 26–30.

[18] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A. Ernst, and Margaret-Anne Storey. 2021. Uncovering the benefits and challenges of continuous integration practices. *arXiv:2103.04251* (2021).

[19] Camilo Escobar-Velásquez, Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2022. Enabling mutant generation for open- and closed-source Android apps. *IEEE Transactions on Software Engineering* 48, 2 (2022), 186–208. https://doi.org/10.1109/TSE.2020.2982638

[20] Erik Flores-García, Goo-Young Kim, Jinho Yang, Magnus Wiktorsson, and Sang Do Noh. 2020. Analyzing the characteristics of digital twin and discrete event simulation in cyber physical systems. In *Advances in Production Management Systems. Towards Smart and Digital Manufacturing*. IFIP Advances in Information and Communication Technology, Vol. 592. Springer, 238–244.

[21] Martin Fowler and Matthew Foemmel. 2000. Continuous Integration. Retrieved November 21, 2021 from https://martinfowler.com/articles/originalContinuousIntegration.html.

[22] Heinz Gall. 2008. Functional safety IEC 61508 / IEC 61511 the impact to certification and the user. In *Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications*. 1027–1031.

[23] Keheliya Gallaba and Shane McIntosh. 2018. Use and misuse of continuous integration features: An empirical study of projects that (mis) use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (2018), 33–50.

[24] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification, and Validation (ICST'18)*. IEEE, Los Alamitos, CA, 1–11.

[25] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'19)*. ACM, New York, NY, 257–267.

[26] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. 385–396.

[27] Smitha Gautham, Athira Varma Jayakumar, Abhi Rajagopala, and Carl Elks. 2021. Realization of a model-based DevOps process for industrial safety critical cyber physical systems. In *Proceedings of the 2021 4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS'21)*. 597–604.

[28] Farshad Gholami, Niousha Attar, Hassan Haghighi, Mojtaba Vahidi Asl, Meysam Valueian, and Saina Mohamadyari. 2018. A classifier-based test oracle for embedded software. In *Proceedings of the 2018 Conference on Real-Time and Embedded Systems and Technologies (RTEST'18)*. 104–111.

[29] Carlos A. González, Mojtaba Varmazyar, Shiva Nejati, Lionel C. Briand, and Yago Isasi. 2018. Enabling model testing of cyber-physical systems. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'18)*. ACM, New York, NY, 176–186.

[30] Leo A. Goodman. 1961. Snowball sampling. *Annals of Mathematical Statistics* 32, 1 (1961), 148–170.

[31] Robert M. Groves, Floyd J. Fowler Jr., Mick P. Couyper, James M. Lepkowski, Eleanor Singer, and Roger Tourangeau. 2009. *Survey Methodology* (2nd ed.). Wiley.

[32] Philipp Helle, Wladimir Schamai, and Carsten Strobel. 2016. Testing of autonomous systems—Challenges and current state-of-the-art. In *Proceedings of the INCOSE International Symposium.* 571–584.

[33] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'17).*

[34] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16).*

[35] David L. Hoover. 2016. Textual variation, text-randomization, and microanalysis. In *Proceedings of the 11th Annual International Conference of the Alliance of Digital Humanities Organizations (DH'16).* 223–225. http://dblp.uni-trier.de/db/conf/dihu/dh2016.html#Hoover16a.

[36] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepCrime: Mutation testing of deep learning systems based on real faults. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21).* 67–78. https://doi.org/10.1145/3460319.3464825

[37] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. 2021. Quality metrics and oracles for autonomous vehicles testing. In *Proceedings of the 2021 14th IEEE Conference on Software Testing, Verification, and Validation (ICST'21).* IEEE, Los Alamitos, CA, 194–204.

[38] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. 2021. Quality metrics and oracles for autonomous vehicles testing. In *Proceedings of the 14th IEEE Conference on Software Testing, Verification, and Validation (ICST'21).* IEEE, Los Alamitos, CA, 194–204. https://doi.org/10.1109/ICST49551.2021.00030

[39] Suzette Johnson, Harry Koehneman, Diane LaFortune, Dean Leffingwell, Stephen Magill, Steve Mayner, Avigail Ofer, Robert Stroud, Anders Wallgren, and Robin Yeman. 2018. *Industrial DevOps: Applying DevOps and Continuous Delivery to Significant Cyber-Physical Systems.* National Academies Press. https://itrevolution.com/book/industrial-devops/.

[40] Nidhi Kalra and Susan Paddock. 2016. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice* 94 (Dec. 2016), 182–193.

[41] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of survey research part 2: Designing a survey. *ACM SIGSOFT Software Engineering Notes* 27, 1 (2002), 18–20.

[42] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of survey research: Part 3: Constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes* 27, 2 (2002), 20–24.

[43] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of survey research part 4: Questionnaire evaluation. *ACM SIGSOFT Software Engineering Notes* 27, 3 (2002), 20–23.

[44] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of survey research: Part 5: Populations and samples. *ACM SIGSOFT Software Engineering Notes* 27, 5 (2002), 17–20.

[45] Klaus Krippendorff. 2004. Reliability in content analysis: Some common misconceptions and recommendations. *Journal of the Royal Statistical Society: Series B (Methodological)* 30, 3 (2004), 411–433.

[46] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19).* ACM, New York, NY, 101–111.

[47] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *Proceedings of the 31st IEEE International Symposium on Software Reliability Engineering (ISSRE'20).* 403–413.

[48] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), Article 202, 29 pages. https://doi.org/10.1145/3428270

[49] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. 2021. When life gives you oranges: Detecting and diagnosing intermittent job failures at Mozilla. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1381–1392.

[50] Antonio Loquercio, Elia Kaufmann, René Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. 2019. Deep drone racing: From simulation to reality with domain randomization. *IEEE Transactions on Robotics* 36, 1 (2019), 1–14.

[51] Ivano Malavolta, Grace Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. 2020. How do you architect your robots? State of the practice and guidelines for ROS-based systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'20).* ACM, New York, NY, 31–40.

[52] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. 2016. Continuous integration applied to software-intensive embedded systems—Problems and experiences. In *Proceedings of the International Conference on Product-Focused Software Process Improvement*. 448–457.

[53] Claudio Menghi, Shiva Nejati, Lionel Briand, and Yago Isasi Parache. 2020. Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. IEEE, Los Alamitos, CA, 372–384.

[54] Vuong Nguyen, Stefan Huber, and Alessio Gambi. 2021. SALVO: Automated generation of diversified tests for self-driving cars from existing maps. In *Proceedings of the 2021 IEEE International Conference on Artificial Intelligence Testing (AITest'21)*. IEEE, Los Alamitos, CA, 128–135.

[55] Helena Holmstrom Olsson, Hiva Alahyari, and Jan Bosch. 2012. Climbing the "Stairway to Heaven'—A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'12)*. IEEE, Los Alamitos, CA, 392–399.

[56] Heejong Park, Arvind Easwaran, and Sidharta Andalam. 2018. Challenges in digital twin development for cyber-physical production systems. In *Cyber Physical Systems. Model-Based Design*. Springer International, Cham, Switzerland, 28–48.

[57] Shari Lawrence Pfleeger and Barbara A. Kitchenham. 2001. Principles of survey research: Part 1: Turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes* 26, 6 (2001), 16–18.

[58] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the vocabulary of flaky tests? In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*. 492–502.

[59] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the vocabulary of flaky tests? In *Proceedings of the 17th International Conference on Mining Software Repositories*. 492–502.

[60] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE, Los Alamitos, CA, 164–175.

[61] Steven P. Reiss. 2007. Automatic code stylizing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 74–83.

[62] Vincenzo Riccio and Paolo Tonella. 2020. Model-based exploration of the frontier of behaviours for deep learning system testing. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. ACM, New York, NY.

[63] Terry Rowlands, Neal Waddell, and Bernard McKenna. 2016. Are we there yet? A technique to determine theoretical saturation. *Journal of Computer Information Systems* 56, 1 (2016), 40–47. http://dblp.uni-trier.de/db/journals/jcis/jcis56.html#RowlandsWM16.

[64] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *Companion Proceedings of the 38th International Conference on Software Engineering (ICSE Companion'16)*. 21–30.

[65] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. 112–122.

[66] Hesham Shokry and Mike Hinchey. 2009. Model-based verification of embedded software. *Computer* 42, 4 (2009), 53–59.

[67] Sebastian Sontges and Matthias Althoff. 2018. Computing the drivable area of autonomous road vehicles in dynamic road scenes. *IEEE Transactions on Intelligent Transportation Systems* 19, 6 (2018), 1855–1866.

[68] Donna Spencer. 2009. *Card Sorting: Designing Usable Categories*. Rosenfeld Media.

[69] Daniel Ståhl and Jan Bosch. 2014. Automated software integration flows in industry: A multiple-case study. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, 54–63.

[70] Daniel Ståhl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 87 (2014), 48–59.

[71] Daniel Ståhl, Kristofer Hallén, and Jan Bosch. 2016. Continuous integration and delivery traceability in industry: Needs and practices. In *Proceedings of the 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'16)*. IEEE, Los Alamitos, CA, 68–72.

[72] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour prediction for autonomous driving systems. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. 359–371.

[73] Sirasak Tepjit, Imre Horváth, and Zoltán Rusák. 2019.The state of framework development for implementing reasoning mechanisms in smart cyber-physical systems: A literature review. *Journal of Computational Design and Engineering* 6, 4 (April 2019), 527–541.

[74] Martin Törngren and Ulf Sellgren. 2018. *Complexity Challenges in Development of Cyber-Physical Systems*. Springer International, Cham, Switzerland, 478–503.

[75] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, New York, NY, 805–816.

[76] Mario Linares Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. 233–244. https://doi.org/10.1145/3106237.3106275

[77] Carmine Vassallo, Sebastian Proksch, Harald Gall, and Massimiliano Di Penta. 2019. Automated reporting of anti-patterns and decay in continuous integration. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE, Los Alamitos, CA.

[78] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. 2020. Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 327–337.

[79] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. 2016. Continuous delivery practices in a large financial organization. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME'16)*. 41–50.

[80] Chunhui Wang, Fabrizio Pastore, and Lionel C. Briand. 2019. Oracles for testing software timeliness with uncertainty. *ACM Transactions on Software Engineering and Methodology* 28, 1 (2019), Article 1, 30 pages. https://doi.org/10.1145/3280987

[81] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. 2021. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 20–31.

[82] Rajaa Vikhram Yohanandhan, Rajvikram Madurai Elavarasan, Premkumar Manoharan, and Lucian Mihet-Popa. 2020. Cyber-physical power system (CPPS): A review on modeling, simulation, and analysis with cyber security applications. *IEEE Access* 8 (2020), 151019–151064.

[83] Fiorella Zampetti, Saghan Mudbhari, Venera Arnaoudova, Massimiliano Di Penta, Sebastiano Panichella, and Giuliano Antoniol. 2022. Using code reviews to automatically configure static analysis tools. *Empirical Software Engineering* 27, 1 (2022), 28. https://doi.org/10.1007/s10664-021-10076-4

[84] Fiorella Zampetti, Damian A. Tamburri, Sebastiano Panichella, Annibale Panichella, Gerardo Canfora, and Massimiliano Di Penta. 2022. Continuous integration and delivery practices for cyber-physical systems: An interview-based study. *ACM Transactions on Software Engineering and Methodology*. Accepted, October 2022. https://doi.org/10.5281/zenodo.6337040

[85] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald C. Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25, 2 (2020), 1095–1135.

[86] Eleni Zapridou, Ezio Bartocci, and Panagiotis Katsaros. 2020. Runtime verification of autonomous driving systems in CARLA. In *Runtime Verification*. Lecture Notes in Computer Science, Vol. 12399. Springer, 172–183.

[87] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. 2021. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21)*. 50–61.

[88] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 385–396.

[89] Celal Ziftci and Diego Cavalcanti. 2020. De-flake your tests: Automatically locating root causes of flaky tests in code at Google. In *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME'20)*. 736–745.

[90] Behrouz Zolfaghari, Reza M. Parizi, Gautam Srivastava, and Yoseph Hailemariam. 2021. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software: Practice and Experience* 51, 5 (2021), 851–867. https://doi.org/10.1002/spe.2929

Received 8 March 2022; revised 27 July 2022; accepted 18 October 2022