Problems and Solutions in Applying Continuous Integration and Delivery to 20 Open-Source Cyber-Physical Systems

Fiorella Zampetti
University of Sannio
Benevento, Italy
fzampetti@unisannio.it

Vittoria Nardone
University of Sannio
Benevento, Italy
vnardone@unisannio.it

Massimiliano Di Penta
University of Sannio
Benevento, Italy
dipenta@unisannio.it

ABSTRACT

Continuous integration and delivery (CI/CD) have been shown to be very useful to improve the quality of software products (e.g., increasing their reliability or maintainability), and their development processes, e.g., by shortening release cycles. Applying CI/CD in the context of Cyber-Physical Systems (CPSs) can be particularly important, given that many of those systems can have safety-critical properties, and given their interaction with hardware or simulators during the development phase. This paper empirically analyzes how CI/CD is enacted in CPSs when considering the context of open-source projects, that often (also) rely on hosted CI/CD solutions, and benefit of an open-source development community. We qualitatively analyze a statistically significant sample of 670 pull requests from 20 open-source CPSs hosted on GitHub, to identify and categorize—also keeping into account catalogs from previous literature—bad practices, challenges, mitigation, and restructuring actions. The study reports and discusses the relationships we found between bad practices/challenges and CI/CD restructuring/mitigation strategies, reporting concrete examples, especially those emerging from the intrinsic complexity of CPSs.

CCS CONCEPTS

• **Software and its engineering** → **Software development process management**; • **Computer systems organization** → *Embedded and cyber-physical systems.*

KEYWORDS

Cyber-Physical Systems, Continuous Integration and Delivery, Pull Requests

ACM Reference Format:
Fiorella Zampetti, Vittoria Nardone, and Massimiliano Di Penta. 2022. Problems and Solutions in Applying Continuous Integration and Delivery to 20 Open-Source Cyber-Physical Systems. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524842.3527948

MSR 2022, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3527948

1 INTRODUCTION

Cyber-Physical Systems (CPSs) are composed of heterogeneous software and hardware units. A peculiar characteristic of CPSs is that they receive inputs from hardware components, e.g., from sensors, and, in turn, send their output to other pieces of hardware, e.g., actuators.

Performing a continuous quality assurance for CPSs can be particularly important, even more than for conventional systems, for several reasons. First of all, CPSs are intrinsically complex, because of the interaction of software components with heterogeneous hardware devices [23, 39]. Second, determining a testing scenario for those systems may imply simulating/mocking "in vitro" the environment [14] in which the system operates, e.g., think about a drone reading inputs from a camera in different weather conditions, reading GPS positions, and controlling rotors to move the vehicle in a given environment.

Setting up a Continuous Integration and Delivery (CI/CD) pipeline to support CPS development could be particularly useful, as it has already been shown for conventional software systems, not only for improving quality assurance but also for reducing development cycles [10, 24, 40, 48].

In industry, CPSs are often developed in closed environments, where, for example, the CI/CD service has access to HiL (Hardware in the Loop) and simulators. At the same time, there has been active development of open-source CPSs, in various domains, ranging from unmanned aerial vehicles to self-driving cars, robotics, or home automation. For these systems, enabling a CI/CD process may require circumventing several challenges, related to the complex and heterogeneous environment, to the extent to which testing activities can be fully automated, as well as to the need for interfacing the system with simulators and HiL.

Similarly to previous studies aimed at identifying software development practices by looking at pull requests' (PRs) discussions [6, 7, 13, 22, 32, 33, 42] we qualitatively analyze PRs to study how developers discuss challenges concerning the application of CI/CD for CPSs and strategies to overcome them.

The goal of this paper is to investigate challenges and bad practices faced when applying CI/CD in the development of CPS open-source projects, as well as specific solutions to the encountered problems. The study has been conducted by performing a qualitative analysis of a statistically significant sample of 670 PRs from 20 open-source projects. The selected open-source projects have topics matching some CPS-related keywords, use at least one CI/CD service, meet specific criteria in terms of the number of commits, PRs, contributors, and are written in C/C++, i.e., the most used languages for CPSs development [36]. Note that we preferred to perform the analysis on a relatively limited set of projects (20),

because this allows us to consider a relatively high number of PRs for each project.

After projects' selection, we identified PRs modifying CI/CD pipelines, building automation scripts, or simulators and/or HiL configurations. Finally, through a hybrid, cooperative card sorting procedure [37], the sample of 670 PRs has been classified in terms of CI/CD bad practices and restructuring actions, as well as challenges and their mitigation.

On the one hand, the study results confirm existing bad practices occurring when setting/evolving CI/CD pipelines for conventional systems, even if some existing problems are exacerbated due to the intrinsic complexity of CPSs. On the other hand, new challenges and related mitigation come up from the study. Specifically, even within a single domain, a CPS must be able to run/be interfaced with multiple/diverse hardware implying "ad-hoc" CI/CD pipeline configurations, e.g., through build matrices for coping with different environments enacted by simulators or HiL.

The data and scripts of our study are publicly available [45].

The paper is organized as follows. Section 2 described the study design and planning. Results are discussed in Section 3, while the threats to their validity are reported in Section 4. Section 5 discusses the related literature, while Section 6 summarizes the main findings of the study, and outlines directions for future work.

2 STUDY DESIGN

The *goal* of this study is to investigate challenges and bad practices arising when setting and evolving CI/CD pipelines of open-source CPSs, and how developers cope with them. The *perspective* is of researchers, interested in studying and improving the development of CPSs. The *context* consists of 20 C/C++ open-source CPSs hosted on GitHub. The paper addresses the following research question:

> *What are the challenges and bad practices occurring when applying CI/CD to CPSs? How are they mitigated/resolved?*

We use the term *challenge* to refer to a demanding task that developers face and wish to overcome during CPS development, e.g., hardware integration. A *mitigation* is a strategy aimed at reducing the severity of the challenge, e.g., limiting the use of HiL in periodic builds to avoid slow continuous builds. With *bad practice*, consistently with previous literature [46], we refer to a bad application of CI/CD principles, such as having a continuous build overcoming the "10-minutes" rule. In the presence of bad practices, developers tend to apply "concrete" *restructuring* (e.g., refactoring) actions to the pipeline configuration files. For instance, developers might simplify a build matrix by removing obsolete environments.

2.1 Context Selection and Data Extraction

The study is based on the qualitative analysis of 670 closed (i.e., merged and unmerged) PRs extracted from 20 CPS-related open-source projects hosted on GitHub.

Previous literature has been studied software development practices by mining PRs. For example, security issues [13, 33], documenting software [6], design decisions [9, 42], software decay [7], or refactoring [32] have been investigated using PRs. Other studies investigated how PRs were used from the perspective of software contributors [21] or integrators [22].

In our work, we focus on PRs because we conjecture that bad practices/challenges together with their related restructuring/mitigation strategies are likely to be discussed by developers, instead of being simply reported as a summary of their changes, i.e., PR title or commit message. By simply analyzing commits and their messages would unlikely allow us to understand the rationale of the changes made, and, above all, the challenges encountered by developers in a given circumstance. Note that, we also consider unmerged PRs because there are cases where a PR discussing a challenge or bad practice together with a possible solution is not merged because it has lower priority with respect to other ongoing activities (e.g., PR #10027 from ARDUPILOT[1] stating: *"GitHub doesn't want to collaborate, continuing on #10589"*). The selected projects are (mainly) written in two different programming languages, i.e., C and C++, which are the most popular for CPS development [36]. We could have considered multiple languages, but then, once again, this would have resulted in very heterogeneous pipelines in terms of used technology, e.g., for compilation, testing, static analysis, or deployment.

To identify CPS-related projects, we leveraged a combination of a preliminary search conducted by using project topics as a query with further refinement. First, we identified a set of GitHub topics relevant for CPS projects: *{automotive, autonomous-driving, autonomous-vehicles, cyber-physical-systems, drone, drones, embedded, embedded-systems, robot, robotics, ros, self-driving-car, self-driving cars}*. As it can be noticed, some topics (i.e., *self-driving-car* and *drone*) appear both as singular and plural, and, since the GitHub topic-based query performs an exact matching, both need to be used. Furthermore, the topics being chosen represent CPS domains (e.g., automotive, drones) rather than components (e.g., sensors). In the end, we performed $13 \cdot 2 = 26$ queries as follows:

```
https://api.github.com/search/repositories?q=topic:
TOPIC+language:LANGUAGE
```

where LANGUAGE can be either *C* or *C++*, and TOPIC one of those reported above. After obtaining the results of the queries, we combined them, removed duplicates, and sorted the projects using the number of forks, which we considered as a proxy of projects' popularity more reliable than stars [8]. Furthermore, in order to obtain projects with (i) enough activity, (ii) active use of PRs, our premier source of information, and (iii) enough contributors so that they generate discussion and benefit enough from the feedback provided by CI/CD, we excluded projects: (i) being forked from others; (ii) not adopting a CI/CD infrastructure; (iii) having less than either 100 commits, 50 PRs, and 50 contributors. The use of CI/CD was checked by combining the automated matching previous work did [44] with a manual check: i.e., by inspecting the repository and searching for files/directories related to CI/CD service configurations, e.g., `.travis.yml`, `.github/workflows`, or `appveyor.yml`.

Rather than sampling PRs from a large set of candidate projects, we prefer to consider a relatively small subset, to avoid sampling zero/one PR from projects, and therefore having results that may not cover different aspects of the studied projects. After the filtering,

[1] https://github.com/ArduPilot/ardupilot/pull/10027. To reach each pull request mentioned in the paper you can simply use the following search query: https://github.com/$OWNER/$REPO-NAME/pull/$PR-NUMBER, where $OWNER/$REPO-NAME is detailed in Table 1 and $PR-NUMBER is reported in the text.

we used Perceval [15], to retrieve the closed PRs from the top 20 projects in the list. As shown in Table 1, the set of projects features 7 CPS applications, 4 simulators, and 9 CPS components, e.g., libraries that can be used to simplify CPS development. As the reader can notice, to make our study more complete, we intentionally did not restrict only to CPSs, but also to other components useful for CPS development.

For each PR, we used the GitHub API to retrieve the set of commits belonging to it, along with the path of the files they impact. It is possible that because of rebasing, some of the initial commits are not visible anymore. However, even in that case, we were able to check the commits that were actually merged. After that, through regular expressions applied to the file names, we restricted the set of PRs to those that contain at least a commit changing: (i) CI/CD service configuration scripts (identified as mentioned before), (ii) build automation scripts (e.g., `Makefiles`), and (iii) files/directories containing simulators or simulator/HiL configurations. The latter were identified through manual analysis of the repositories. For instance, in RUSEFI there are two specific directories dealing with `hardware` and `simulator`, while in PX4-AUTOPILOT under the Tools folder there are the third-party simulators forked in the current version of the project.

Table 1: # of PRs in the sample scattered across the 20 CPS open-source projects

First Round - Projects	Category	# closed	# filtered	# sampled
PX4/PX4-Autopilot	CPS	11,598	2,881	149
paparazzi/paparazzi	CPS	2,035	1,827	94
arduPilot/ardupilot	CPS	12,953	759	39
cyberbotics/webots	Simulator	2,678	695	36
carla-simulator/carla	Simulator	1,069	282	15
cleanflight/cleanflight	CPS	1,385	213	11
cartographer-project/cartographer	CPS Component	1,277	141	7
nasa/fprime	CPS Component	522	138	7
mavlink/mavros	CPS Component	410	74	4
bulletphysics/bullet3	CPS Component	1,686	36	2
TOTAL FIRST ROUND	–	35,604	7,046	364
Second Round - Projects	**Category**	**# closed**	**# filtered**	**# sampled**
rusefi/rusefi	CPS	2,016	454	93
ros-planning/navigation2	CPS	1,567	385	79
dartsim/dart	CPS Component	962	253	52
PX4/PX4-SITL_gazebo	Simulator	550	157	32
CopterExpress/clover	CPS Component	337	75	15
simbody/simbody	Simualtor	411	66	13
ArduPilot/apm_planner	CPS	385	52	11
linux-can/can-utils	CPS Component	170	26	5
UAVCAN/libcanard	CPS Component	106	19	4
ompl/ompl	CPS Component	152	10	2
TOTAL SECOND ROUND	–	6,620	1,497	306
OVERALL	–	**42,224**	**8,543**	**670**

2.2 Qualitative analysis methodology

To address our research question, we manually analyzed a sample of 670 PRs, through a hybrid card sorting strategy [37], since we started from a predefined set of bad practices and restructuring actions arising when setting/evolving pipeline for traditional systems. The analysis has been conducted in two different rounds. We started by sampling and analyzing PRs from the top ten-ranked projects to conduct a self-contained PR tagging to define a set of categories (bad practices, challenges, mitigation strategies, and restructuring actions) as much complete as possible. At the end of the first round, we realized that, differently from the study by Zampetti et al. [46], in the CPS context developers also face challenges when setting/evolving their pipelines. For this reason, we performed a second round looking at PRs extracted from a completely different set of projects, i.e., the subsequent 10 projects in the ranked list. In other words, with the second round, we wanted to see the extent to which the initial set of categories generalized on unseen data.

In both rounds, we performed a random-stratified sampling (strata are the projects, i.e., PRs were sampled proportionally in each project) over candidate PRs from the 10 projects included in each round, obtaining a sample size of 364 and 306, respectively (confidence interval ± 5%, confidence level 95%), as shown in Table 1. As the reader can notice, the second set of projects has an overall number of candidate PRs way smaller than the first one. This is likely because projects with more forks naturally receive more PRs from external contributors, but it is especially due to the very high number of PRs some projects in the top-10 have, in particular, PX4-AUTOPILOT and PAPARAZZI.

The manual analysis was performed by two independent annotators, i.e., the first two authors of this study, by applying the following procedure. First, each annotator determined whether a PR discussed CI/CD challenges or bad practices, and whether or not they are CPS-specific. Specifically, we consider a challenge/bad practice to be CPS-specific when it is directly related to the usage/setting of HiL or simulators in the pipeline, or whether it concerns a behavior related with a CPS interaction, e.g., a flakiness resulting from sensor data or, in general from CPS interaction. As an example, consider PR #14228 from PX4-AUTOPILOT where developers are struggling with different behaviors between real hardware and data coming from simulators, i.e., "gazebo sensor rates are very different from real hardware". Then, each PR has been further categorized with a label describing the challenge/bad practice, as well as the mitigation strategy adopted to overcome it. Note that the labeling procedure started from a predefined list of categories related to (i) bad practices in setting, using, and maintaining CI pipeline [46], and (ii) restructuring actions applied during CI/CD pipelines evolution [44].

The PR labeling has been performed using an online spreadsheet where the annotators could use drop-down menus to select previously defined categories, or add a new one when those did not fit. At the end of each round, an open discussion was performed by adding a third annotator, i.e., the third author of the study, by checking all the PRs for which there were disagreements among the two annotators (152 out of 364 and 88 out of 306 in the two rounds, respectively). Furthermore, we also discussed PRs having agreements and *yes* labels by both two annotators (52 and 27 in the two rounds respectively). Based on the labeling procedure, we could have different types of disagreements: (i) one annotator labeled the PR as discussing an issue, while the other did not find it; (ii) the two annotators disagree regarding whether the discussed problem is CPS-specific or not; (iii) the annotators assigned different challenges; and (iv) the annotators disagreed about the mitigation strategy.

In the first round, the annotators agreed in 73% of the cases on whether a PR discusses a challenge or bad practice (80% in the

Table 2: CI/CD setting of the 20 CPSs open-source projects in the study

Project	Description (Type)	CI/CD Fram.	Containers	3rd-party Sim.	HiL	Build Environment
PX4-Autopilot	Drone flight controller (CPS)	AppVeyor GitHub Jenkins Travis-CI	☑	☑	☑	HW-based
Paparazzi	Unmanned air vehicles system (CPS)	Travis-CI	☑	☑		Compilers-based
Ardupilot	Vehicle autopilot systems (CPS)	AppVeyor GitHub Semaphore	☑	☑	☑	HW/Execution Env-based
Webots	Robot (Simulator)	GitHub				OS/ROS-based
Carla	Autonomous driving (Simulator)	GitHub Jenkins Travis-CI		☑		Tasks-based
Cleanflight	Flight controller (CPS)	Azure GitHub Travis-CI			☑	Compilers-based
Cartographer	Localization and mapping system (CPS Component)	AppVeyor GitHub Travis-CI	☑			OS-based
Fprime	Framework for spaceflight applications (CPS Component)	GitHub				Tasks-based
Mavros	Ground control station communication (CPS Component)	GitHub		☑		OS/Language-based
Bullet3	Physics/collision simulation (CPS Component)	AppVeyor GitHub				
Rusefi	Control unit engine (CPS)	GitHub		☑	☑	HW-based
Navigation2	Robot navigation system (CPS)	Circle-CI GitHub	☑	☑		
Dart	Animation/robotics toolkit (CPS Component)	GitHub	☑			OS-based
PX4-SITL_gazebo	Flight (Simulator)	GitHub	☑	☑	☑	OS/ROS-based
Clover	Autonomous drone control (CPS Component)	GitHub	☑			ROS-based
Simbody	Dynamics/physics (Simulator)	AppVeyor GitHub Travis-CI				OS-based
Apm_planner	Ground station application (CPS)	Travis-CI	☑	☑		OS-based
Can-utils	CAN bus utilities (CPS Component)	GitHub Travis-CI				Tasks-based
Libcanard	Intra-vehicular communication (CPS Component)	GitHub	☑			Compilers-based
Ompl	Motion planning library (CPS Component)	AppVeyor Travis-CI	☑			OS-based

second round). While these percentages seem high, it is possible that the annotators agreed by chance. Therefore, we computed the Cohen's k [12], which resulted to be 0.4 (fair to moderate) in the first round, and raised up to 0.53 (moderate) in the second round. Then, we computed the Krippendorff's α [26] inter-rater agreement in terms of whether or not the challenge/bad practice is CPS-specific, and in terms of kind of bad practice, challenge, restructuring action and mitigation. Krippendorff's α was used as for such fields the labeling could be incomplete (given PR, an annotator could have specified the bad practice, while the other none). For what concerns the specificity to the CPS domain, we obtained $\alpha = 0.68$ (first round) and $\alpha = 0.67$ (second round), both considered acceptable agreements. Also for the bad practices we obtained an acceptable agreement in both rounds ($\alpha = 0.73$ and $\alpha = 0.84$), while the agreement is slightly below the threshold of the acceptable agreements in terms of challenges ($\alpha = 0.61$ and

$\alpha = 0.63$, respectively). The latter is lower because for the challenges we did not start from any previously defined set of categories. For the restructuring actions in both rounds, we found an acceptable α (> 0.73), while we were not able to compute the Krippendorff's α [26] for the mitigation strategies, since we have few data points in both rounds, i.e., 27 and 25, respectively. For that, we avoided agreement by chance by reviewing every single case.

In the first round, annotators used 22 bad practices, and 14 restructuring actions from the previously defined categorizations [44, 46], and added 7 unseen bad practices and 2 restructuring actions. Furthermore, they introduced 17 CPS-specific challenges, addressed with 8 different mitigation strategies. During the second round, instead, the annotators (i) reused 31 bad practices, 15 restructuring actions, 10 challenges and 4 mitigation strategies, and (ii) added one new bad practice, one restructuring action, one challenge and one mitigation.

To sum up, as outcome of the two rounds, we produced a list of: (i) 34 bad practices, among the 79 elicited by previous work [46] that occurred in the analyzed CPS projects, with 8 additional ones; (ii) 18 challenges faced when applying CI/CD to CPSs; (iii) 9 mitigation strategies for challenges; and (iv) 16 restructuring actions to the CI/CD pipelines among the 34 from previous work [44], with 3 additional ones.

3 STUDY RESULTS

This section discusses the findings of the study detailed in Section 2. We start by discussing the CI/CD pipeline setting for the 20 studied projects. Then, we discuss challenges and related mitigation, and, finally, bad practices with restructuring actions. For the latter, we report and discuss only relations occurring more than once in our sample (others are in the replication package), and emphasize the discussion of CPS-specific problems. Note that, while all the considered PRs and discussed challenges relate to changes to CI/CD configuration, some may be valid to CPSs in general, although they will inevitably have a direct impact on the application of CI/CD, especially because they make its automation harder and its feedback less reliable.

3.1 CI/CD pipelines of the studied projects

Table 2 provides an overview of the projects' pipelines as it is observed on their main branch on GitHub to date[2].

Half of the projects rely on more than one CI/CD framework. In some cases, e.g., PX4/Autopilot and Ardupilot, different frameworks are used for local tests on HiL, and for tests on simulators on the cloud. Also, we found cases where some frameworks are not updated anymore in the projects (refer to the ones having both Travis-CI and GitHub), and cases where developers are migrating to a different framework. Unsurprisingly, 17 out of 20 projects adopt the GitHub CI/CD framework.

11 projects use containers in the pipeline for different purposes. For instance, PX4-Autopilot, Ardupilot, cartographer, and ompl use Docker containers to deal with different operating systems, clover has different Ubuntu images dealing with different `raspberry-pi` models, while paparazzi is the only one relying on virtual machines.

[2]Last accessed at January 18, 2022

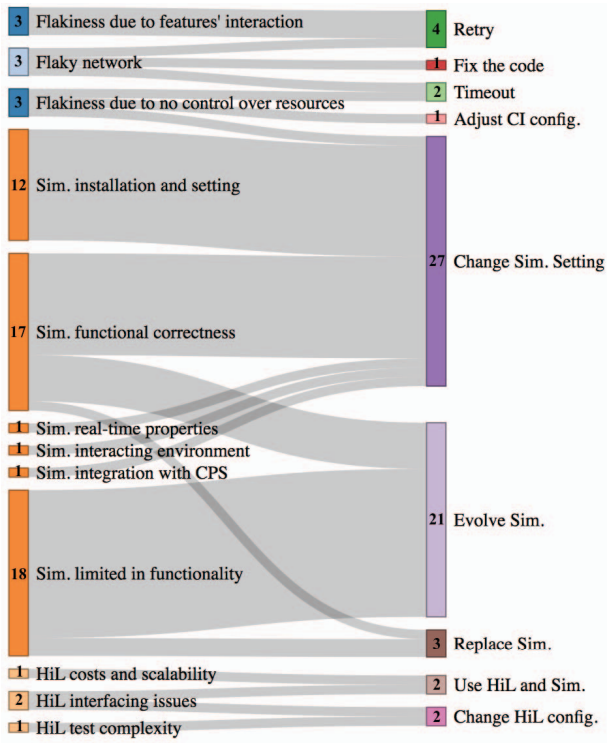

Figure 1: Relations between Challenges and Mitigations

Nine projects rely on third-party simulators. Note: a simulator may, in turn, rely on other simulators for specific aspects, i.e., carla (which is a car simulator) relies on other simulators, in this case, a traffic simulator PTV-vissim [3], whereas PX4-SITL_gazebo is a set of simulator plugins that, to be tested, need to be integrated with the Gazebo robot simulator [2].

In some cases, the pipeline includes running the software on HiL (sixth column in Table 2). Only a quarter of the projects have it, likely due to the open-source domain and the prevalence of hosted, cloud-based CI/CD services.

All projects, except navigation2, have multiple build environments in the same pipeline. In some cases, the purpose is to run CI on different operating systems, e.g., ROS (Robot Operating System) distributions, or simply Linux/MacOS/Windows versions, while others customize environments based on the compilers or the type of tasks included in the pipeline. Moreover, multiple environments are used to deal with multiple hardware devices, as for PX4-Autopilot where there is a build environment for each board.

3.2 Challenges and Mitigation Strategies

Figure 1 shows the challenges developers face when setting and evolving a CI/CD pipeline for CPS development, together with the mitigation strategies adopted to overcome them. To simplify the discussion, the challenges have been grouped into three different categories, each one related to a specific aspect of the CI/CD pipeline—flaky behavior, simulators, and HiL. For each category,

we report a brief description of the identified challenges, and by using qualitative examples, we highlight the strategies adopted to deal with them.

3.2.1 Flaky behavior. This category accounts for three different root causes that lead to non-determinism in the build execution.

Flakiness due to no control over resources, similarly to what occurs in conventional systems relying on CI/CD, may manifest when the pipeline owner has no control over external resources included in the CI/CD process, such as the load on the server-side, or the inappropriate initialization of the server used by the CI/CD process, e.g., PR #12373 in PX4-Autopilot stating: *"Sometimes Jenkins misses the initial boot after upload and gets stuck"*. As expected, there are different ***mitigation strategies*** used to deal with the problem: (i) use timeout, (ii) modify the configuration of the pipeline, and (iii) change the simulators' setting so that it is possible to recognize issues dealing with the impossibility to reach the simulators within the CI/CD process. An interesting example comes from PR #576 in PX4-SITL_gazebo, where developers had to modify the configuration so that it was possible to discriminate SITL and firmware tests, to avoid possible interference between them, leading to *"intermittent failures"*.

We found three different PRs discussing **Flaky Network**. Consider PR #2234 from paparazzi where *"2 seconds is a bit to[o] short in some cases when the router takes longer to reply"* where developers ended up with increasing the timeout, i.e., *"5 seconds gives much better results."* In other cases the ***mitigation strategies*** were (i) using retry, or (ii) identifying the root cause of the problem and fixing it, as in PR #2170 from navigation2 where it was required *"[u]pdating warning that occasionally crashes test on rclpy QoS updates"*. While flakiness due to network issues also occurs in conventional software, in CPSs the interaction between different devices may make things worse.

Finally, since CPSs are systems of systems, it is possible to have **flakiness due to features' interaction**, due to the presence of a complex interacting environment. Interestingly, PR #598 reports a discussion where developers struggle to have 100% of test passing in CI, due to a high level of interaction between different features under test, i.e., *"system test"*. The ***mitigation strategy*** used deals with retry, e.g., changing the number of retries, as it happens in PR #984 from Navigation2 for *"debug[ging] stability issues"*.

3.2.2 Simulators. This category groups seven challenges related to simulator issues and limitations. One out of seven does not come from PRs where developers also provide a ***mitigation strategy***, i.e., **Choose among different simulators**. For instance, PR #14539 from PX4-Autopilot discusses and compares the features provided by two different simulators to decide whether or not to change the one their pipeline relies on. Specifically, PX4-Autopilot relies on `Gazebo`, however *"FlightGear has better support for modeling of rotor-craft than the current PX4's mainstream simulator Gazebo."*

Simulators limited in functionality is mentioned in 20 PRs of our sample. This challenge can be due to the presence of a complex environment that must be simulated, or to the use of a third-party simulator still under development. As ***mitigation strategies*** developers tend to (i) add the missing functionality, i.e., evolve the simulator, or (ii) replace the simulator with a better one. As an example of (i), in PX4-Autopilot, we found a PR (#7235) where it was needed to *"[s]upport multi uav simulation in SITL with Gazebo+MAVROS"*, requiring developers to write code for the missing functionality. As for (ii), PR #11835 in ardupilot discusses the need to replace the Morse simulator with AirSim since the previous one was not properly maintained. Note that, ardupilot does not use the simulators as external libraries while importing them into its code base and evolves the simulators based on the needs.

Simulators' functional correctness includes cases where wrong assumptions about the system/device to simulate, negatively impact the simulator's correctness. This can be fixed with three ***mitigation strategies***: (i) by simply changing the simulators' settings, (ii) by evolving/patching the simulator, or (iii) by replacing it with a different one. As an example of changing settings, PR #13060 from ardupilot modifies the parameters set in the AirSim configuration to fix latitude/longitude accuracy: *"fixed accuracy of lat/lon in AirSim"*. In terms of simulator fixes, PR #2128 in paparazzi fixed measurements issues in the simulator since *"Gazebo's ... did not seem to take the ardrone's velocity_decay (drag) into account, instead it measured the drone's acceleration without drag which resulted in incorrect measurements in flight"*. Finally, as regards simulator replacement, in navigation2 there is PR #2037 which reverts a previous change done to patch a bug in the simulator being used. In other words, in a previous PR developers replaced the simulators with a different one, since they were waiting for a concrete fix in the original simulator. Once the simulator was fixed, they replaced the simulator in use.

Simulator installation and setting includes problems related to the way developers configure the simulator to use in the pipeline. Obviously, among the ***mitigation strategies***, the one being used deals with changing the simulator configuration. As an example, in PR #15206 from PX4-Autopilot, developers point out the occurrence of silent failures due to a wrong strategy for cleaning the server in which the simulator runs— *"When running gazebo SITL simulation multiple times, gzserver will sometimes silently fail. This is due to the fact that gzserver didn't exit cleanly from the previous session."* PR #12781 of the same project, instead, discusses how to run the simulator detached from its GUI (i.e., in headless mode) to integrate it with a CI/CD pipeline on a cloud infrastructure.

Simulator real-time properties includes problems encountered by developers when real-time requirements for the system have to be met. Only one PR belonging to this category provides a ***mitigation strategy*** that tweaks the simulator's setting. Specifically, PR #963 from paparazzi discussed the need for simulating `sys_time` in `OCaml`. While adding this feature, developers realized that *"unfortunately, the fixed freq[uency] of the ocaml sim has many stupid effects if you run at a different frequency."* The latter implies a change in the simulator set to *"redefine and use"* the `PERIODIC_FREQUENCY` value.

Simulator interacting environment includes problems faced when the simulator interacts with a too complex environment that must be simulated as well. Among the six PRs reporting this challenge, only one (#2473) in navigation2 also discusses the ***mitigation strategy***, that is, similarly to the previous cases, to modify the simulator's setting. Developers realized that to have

a properly working simulator, *"gazebo should be started with both libgazebo_ros_init.so and libgazebo_ros_factory.so"*.

Last, but not least, **Simulators integration with CPS** includes problems developers encounter when they need to interface the simulators with real devices in the pipeline. Among the two PRs belonging to this category, only PR # 237 from CLOVER has a ***mitigation strategy***. In this case, the PX4 firmware was just copied into the CLOVER project and adapted to the project's needs. While a CLOVER-specific version of PX4 is needed, developers suggest having it in a separate workspace, i.e., available as a *ROS* node. However, this configuration could make the integration difficult for the users, and developers suggest the use of containers or virtual machines.

3.2.3 HiL. This category includes six challenges related to issues and limitations of adopting HiL within the pipeline. However, as shown in Figure 1, only for three of them there is a PR where developers also provide the mitigation adopted to deal with them. **Lack of HiL specific smoke test** mentioned in PR #12282 from PX4-Autopilot, points out a problem with not having a smoke test procedure on the HiL-side—*"Jenkins HIL test run various commands to inspect the system"*. Smoke tests aim at testing the pipeline itself, and for CPSs it also concerns testing the proper interaction with HiL. Also in PX4-Autopilot, PR #16396 describes parallelization problems when trying to use different real devices simultaneously (i.e., **Parallelization and Resource Blockage**). Finally, when using both simulators and HiL within the pipeline, it is possible to see **Testing Discrepancies between SiL and HiL**[3]. For instance, PR #193 from CLOVER states that developers *"could [not] figure out why the tests are failing, though: they do [not][fail on our separate x86_64 jobs and they pass on real hardware.*

Focusing on PRs where mitigation strategies are described, there are cases where testing on HiL is constrained by the high cost and lack of scalability of the hardware devices (i.e., **HiL costs and scalability**). Whenever possible, as a ***mitigation strategy***, developers use staged builds where they include simulators and HiL to reach a good confidence level about the correct behavior of the system under development (see PR #1496 from CLEANFLIGHT).

A different challenge deals with **HiL test complexity**, meaning that testing on HiL is very demanding to put in place, due to the presence of a huge number of different devices to consider. Only one out of six PRs describing this challenge also provides a possible ***mitigation strategy*** that implies changing the HiL configuration used in the pipeline. Specifically, PR #13722 from PX4-Autopilot faces the challenge when a user tried to build a drone using `raspberry pi` with a `navio2` connected to the test rack. One of the project owners reports that *"Hardware drivers cannot be added to automatic testing because the RPi platform is built up by a lot of different sensor solutions in most cases. Things that CI can do are limited to common modules."*

Last, but not least, **HiL interfacing issues** includes 5 PRs, with two also reporting a ***mitigation strategy***. In PR #171 from APM_PLANNER, developers had problems interfacing with the device's serial port and solved it by using the *libudev* API to introspect devices available in the local system. Instead, PR #14156 from PX4-Autopilot discusses a problem related to *"matching between driver and configured device"* that is a *"bit tricky for drivers supporting multiple devices, like ms5611 and ms5607 that do auto-detection. It works, but both sides (driver+board config) need to use the same IDs."* In this case, no clear solution was found. PR #14054 from PX4-Autopilot, instead, mentions the need of relying on mocking to speed up the builds. Specifically, developers create a *"mockup optical flow model, that computes a flow measurement without rendering an image"* to speed up the *"CI SITL tests"*.

3.2.4 Others. This category accounts for two challenges that cannot fit any previous categories described, and for which no mitigation strategy was proposed. Very often developers deal with **Limited software and/or hardware resources** influencing the type of execution environment used in the pipeline. An example is described in PR #17211 from PX4-Autopilot where *"the main stack of the tests is quite large, so it [is] increasingly problematic on older boards that do [not] have enough memory to run both the tests runner and allocate a new task or thread for their test activity.* Also, developers might face issues related to the **presence of complex, non-functional requirements**, e.g., real-time operating systems, that might constrain the pipeline settings. For instance, in PR #3170 from PX4-Autopilot developers struggle with *"improving the portability of the firmware to new boards."*

3.3 Bad Practices and Restructuring Actions

Figure 2 shows how developers address the bad practices through restructuring actions, as found in our sample. We only report and detail (with qualitative examples) relations occurring more than once, while you can find all the occurrences in our replication package [45]. Since for bad practices and restructuring actions we started with what already known from previous literature [44, 46], Figure 2 highlights with a (*) bad practices and restructuring actions not found in previous literature.

3.3.1 Infrastructure Choices. We found two different bad practices dealing with poor software choices when setting the pipeline. **Use an unsuitable tool/plug-in** relates to the inappropriate setting of the cache, or to the adoption of a tool increasing the build execution time. The possible ***restructuring actions*** in this case are: (i) change the cache configuration, (ii) move to containerization, and (iii) replace the tool/plug-in with a suitable one (see Figure 2). As an example of (i), PR #4208 from ARDUPILOT discusses the impact the cache size has on compilation time: *"how ccache makes things faster is retaining some info between builds when we change targets. Or, do we clean between each target and thus don't need ccache anyway?"* and decide to increase it. In the same project, PR #3228, instead, describes the advantages of adopting the container-based infrastructure rather than virtual machines, as a more lightweight solution also allowing caching facilities, i.e., *"The biggest advantage of using the container-based one is the ability to cache a) things we need to download and b) intermediary build steps by using ccache"*.

Inappropriate CI/CD framework includes cases where developers deal with limitations in the CI/CD infrastructure being used, e.g., a limited number of parallel tasks is allowed, or difficulty to automate certain tasks. For instance, PR #15463 from ARDUPILOT, as a ***restructuring action***, replaces the CI/CD infrastructure for running a specific task, i.e., from Travis-CI to GitHub Actions,

[3] **SiL** is acronym of **S**oftware **i**n the **L**oop — **HiL** is acronym of **H**ardware **i**n the **L**oop

Figure 2: Relations between Bad Practices and Restructuring

mentioning as an advantage the build execution time reduction due to the possibility of having *"20 tasks in parallel"*, other than a *"fast boot"*.

3.3.2 Build Initialization. This category has one bad practice, i.e., **Inappropriate build environment clean-up** addressed using five possible ***restructuring actions***: (i) change cache configuration; (ii) clean-up build matrix aiming at improving performance; (iii) reorder build steps; (iv) update checks in the build process; and, (v) modify the triggering strategy. While such changes apply for conventional systems too, the complexity of CPSs may impact the build duration in a way that the build process needs to be properly reconfigured. To this regard, we found cases where, by adding a check in the build process, developers modify what has to be executed based on the type of change (e.g., PR #3478 from webots), or else modify the triggering strategy by adopting incremental builds (e.g., PR #4458 from PX4-Autopilot).

3.3.3 Build Process Definition. This category features two bad practices dealing with how jobs are used in the CI/CD process: (i) **Use wide and in-cohesive build jobs** and (ii) **Use monolithic builds**, mainly fixed using as ***restructuring actions***, the splitting of build scripts/jobs, or simplifying the build matrix. Focusing the attention on the CPS context, we found PR #9692 from PX4-Autopilot where developers introduces different jobs each based on a specific task, and defined a job for separating the software-in-the-loop (SiL) tests from other types of tests executed in the pipeline. On the contrary, developers from ardupilot combined *"SITL and unit-test tasks ... in the same instance"* once realized that they *"spent more time spooling up/installing tooling then we did doing either of these tasks"* (PR #13265).

3.3.4 Build Execution. This category groups four bad practices, one not part of previously proposed bad practices for conventional systems [46], i.e., **Include unneeded environment**, which is obviously addressed with ***restructuring actions*** that (i) remove unneeded environments (used 4 times), or (ii) clean-up the build matrix (used one time). In other words, and this is particularly true for CPSs, the variety of devices, firmware versions, OS versions on which a system needs to be tested change over time, and keeping unneeded ones just slows down the build. For instance, PR #1507 from rusefi removes a duplicated environment, since developers realized that *"The workflows for console with Java version 8 and the one intended for Java 11 are identical"*.

For what concerns the bad practices already known from previous literature [46], we found cases where CI/CD parallelization is inappropriately used, and cases where build steps are inappropriately ordered, leading to a pipeline that is not able to find bugs following the "Fast Feedback" principle (see Figure 2). Furthermore, while previous literature encourages parallelization for what possible [16, 46], developers must pay careful attention to dependencies among steps. For instance, PR #1094 from cartographer mentions: *"the tests in cartographer/cloud cannot run concurrently with each other, use tags* `exclusive` *to say that they should not be run at the same time as other tests."*

3.3.5 Build Triggering. We found one bad practice in this category, i.e., **Poor triggering strategy**, and, as suggested by its name, it includes cases where developers apply a wrong build triggering strategy. For instance, PR #2588 from rusefi discusses the need of having the possibility to run the build manually for *"pinout regeneration"*, so that is possible to create an updated version of the artifacts based on developers' needs. As ***restructuring action***, developers enable manual builds in the GitHub workflow action by adding the `workflow_dispatch` option in the `.yml` files.

3.3.6 Build Dependency Management. This category groups two bad practices related to the strategy adopted for dependencies' handling that might lead to unnecessary build failures: **Inappropriate cache handling**, and **Include unneeded dependencies**. The ***restructuring actions*** we found are mainly related to change (i) the cache configuration, or (ii) the dependency installation policy. As an example of the latter, PR #924 from mavros reports the need to *"blacklist HIL for APM since it is not relevant"* since there is no support for HiL.

3.3.7 Build Outcome. In the presence of a build resulting in a complex and heterogeneous set of warnings—and for CPSs they may come from the system itself, from simulators, HiL interfacing, etc.—developers need to carefully ponder how they influence the build outcome, i.e., passed or failed. Four out of eight PRs belonging to **Green build with errors/failures** discusses the need

for having warnings failing a build if it is triggered before a new release of the code, e.g., PR #1629 from DART stating: *"Treat warnings as errors on release mode"*. As regards **Do not fail a build as soon as a failure/error is encountered**, PR #11529 from ARDUPILOT stresses the importance of the "Fast-Feedback" principle, i.e., *"exit on panic so we don't waste time waiting around"*, fixed adding a check in the build process definition.

3.3.8 Build Output. Other than determining the build outcome, the output of various tools also influences the content of build logs. In this category we found two (opposite) bad practices: **Miss info in build logs**, and **Build log with verbose info**, mainly addressed using, as possible ***restructuring actions***, (i) improving the readability of the build log (used in the majority of cases), or (ii) changing how the build outcome is determined. PR #1223 from DART discusses a problem with build logs not clearly reporting that some components are failing due to external dependencies being missing. In this case, instead of having a failed build, developers ask to add a configuration error being reported in the build log.

3.3.9 Build Duration. For what concerns the bad practice **Include unneeded tasks**, in our sample, we found, as possible ***restructuring actions***: (i) remove unneeded environments/scripts/tasks, and (ii) update checks in the build process. For instance, PR #16191 from PX4-AUTOPILOT reports a discussion about the presence of *hundreds of [unneeded] commands executed across all the boards* fixed by updating the checks in the `Jenkinsfile-hardware` workflow. For **Build too long**, developers tend to use the same restructuring actions as before, plus replace tools/methods for accomplishing a specific task. From a CPS perspective, we found a PR (#205) from NAVIGATION2 where developers found that *"when building from source it takes too long (> 50 minutes) to build ROS2"*, and conclude that *"if there's a daily build (or at least weekly) that could generate a ros2:latest docker image for us to pull and use in our testing."* In other words, whenever possible, finding an out-of-box Docker image instead of building a complex environment make the build much faster.

3.3.10 Build Maintainability. In this category, we found two bad practices dealing with poor build maintainability and portability. For instance, if a pipeline configuration is highly-customized over a specific environment, it is not easy to make it working in a completely different one, and rework is needed to let the system operate in multiple OSs. **Cloned build configurations** has as possible ***restructuring actions*** (i) remove unneeded environments/scripts/tasks, and (ii) use matrix expansion feature or list job configuration explicitly. For instance, PR #2843 from RUSEFI suggests to removes two duplicated build configurations dealing with different hardware devices, i.e., boards, while introducing a single configuration and relying on the matrix expansion feature to specify the different boards (*"using a matrix for different boards/machines"*). Another bad practice, not specific to CPSs yet occurred in our sample, that is **Poor naming of build jobs** is related to naming conventions, e.g., see PR #3056 from RUSEFI.

3.3.11 Security. In the original work about CI/CD bad practices [46], this category features one bad practice related to authentication credentials in clear in the CI/CD scripts. Further security issues in infrastructure-as-code were defined by Rahman et al. [34]. In our study, instead, we found a new bad practice (not considered in the above works): **Use sudo for accessing Docker** with only one ***restructuring action*** related to Introducing/Removing `sudo` in commands. An example is PR #1318 from CARLA reporting: "*It is bad practice to use sudo for accessing docker.*." The Docker daemon usually requires root privileges. This means that a container can easily alter the host file system without any restriction. By exploiting a privilege escalation attack, a malicious user may gain root user access to the host. The best way to prevent this kind of attack is to run processes with a non-root user or create a docker group composed only of trusted users [1].

4 THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation. The biggest challenge is the extent to which our interpretation of PR discussions reflects the challenges encountered by developers. We mitigated this challenge by (i) having multiple annotators to assess the PRs, and (ii) inspecting not only discussions but also the changes made to CI/CD-related artifacts. That being said, it would be desirable to complement the PR-based study with further studies, e.g., by surveying developers. Measurement errors may be due to subjectiveness and difficulties in the manual classification of PRs. This threat has been mitigated by having multiple, independent annotators, analyzing the inter-rater agreement, and reviewing every single case (including agreement cases) where a PR contained the annotation of any between bad practices, challenges, mitigation strategies, and restructurings. This means that, in the worst case, we could have missed some relevant cases, although we have confidence in the classification of the considered ones. The analysis of changes could be affected by history rewriting, however when a change was positively considered in a PR discussion, then its commits (or a cherry-picked subset of them) were accessible.

Threats to *internal validity* are mainly about the extent to which we can claim the cause-effect relationships between challenges/bad practices and mitigation/restructuring actions. In our study, we went beyond the simple "co-occurrences", but we determined, by looking at the developers' discussion, whether a given change was actually performed with the purpose of addressing a challenge/bad practice documented in the PR discussion.

Threats to *conclusion validity* may depend on the sampling strategy being used for projects and for PRs. As for the projects, we restricted our attention to projects with non-empty topics on GitHub, yet there could be other projects, belonging to the same topic, not properly tagged by the authors on GitHub. For what concerns the software artifacts used for identifying challenges/bad practices and related mitigation/restructuring actions, similarly to previous studies aimed at studying software development practices [6, 7, 9, 13, 32, 33, 42], we looked at PR discussions, even if we are aware that it is still possible to extract developer's intent from commit messages. Finally, for PR sampling, as explained, the two rounds were completely independent, therefore we did not compute a single sampling. However, because of that, since the last 10 projects had fewer PRs than the first one, the sampling was more intense for them.

Threats to *external validity* are about the generalization of our findings. The study is intentionally small in terms of the number of projects, and therefore our results may not generalize beyond those systems. As explained in the introduction and in Section 2, the selection of a limited number of projects has been necessary to sample enough PRs per project. Moreover, the selected topics may not exhaustively cover the universe of CPS projects on GitHub, indeed we only consider topics representative of CPS domains rather than components, e.g., sensors. Nevertheless, we agree that a wider selection of topics could have been resulted in a different set of projects. Furthermore, the 670 PRs can be considered representative of the chosen projects, covering different CPS domains in the open-source, while they will not generalize to the universe of CPSs. Also, the study is focused on C/C++ projects and open-source projects. Further research is required to study other languages (less prevalent than C/C++ for CPSs [36]), and above all closed-source, for which, however, a different study methodology such as interviews and ethnography would be required.

5 RELATED WORK

This section discusses related literature about CI/CD barriers and bad practices and challenges in CPS development.

5.1 CI/CD barriers and bad practices

Through interviews, Hilton et al. [24] investigated barriers encountered by developers when moving toward CI. The studied barriers are mainly related to quality assurance, security, and flexibility. Olsson et al. [31], instead, looked at barriers faced by companies moving towards CD, i.e., complexity of the deployment environment, need to achieve timely delivery, and lack of a complete overview of all the development projects. Previous literature also defined bad practices arising from wrong or sub-optimal application of the CI/CD process. Specifically, Duvall [17] defined CI/CD antipatterns, while Zampetti et al. [46] empirically elicited bad practices from interviews and Stack Overflow posts. Finally, Zampetti et al. [44] defined, by mining CI/CD configuration changes, 34 pipeline restructuring actions aiming at improving extra-functional properties and/or changing the pipeline's behavior.

The aforementioned work focuses on conventional CI/CD, whereas our study is specific to CPSs and highlights challenges in that context. While we used the bad practices [46] and restructuring actions [44] of previous literature as a starting point, our work goes beyond that because (i) Zampetti et al. [46] inferred bad practices from Stack Overflow discussions and interviews, while we identify them in actual projects, (ii) both bad practices and restructuring actions have been extended (and specialized) to new specific problems and solutions, (iii) we also identify challenges and mitigation, and, last but not least, (iv) we relate bad practices to restructuring actions, and challenges to mitigation.

5.2 Challenges in CPS development

As pointed out by several studies, CPS development is much more challenging than conventional software.

Törngren et al. [39] investigated general challenges related to CPS design focusing on the complexity of the environment in which these systems operate. We study how the complexity of CPSs is dealt when setting up CI/CD pipelines, for example, by leveraging virtual machines and containers, and how simulators are interfaced with the pipeline.

Mårtensson et al. [30] identified factors to consider when applying CI to software-intensive embedded systems such as complexity of user scenarios, compliance to standards, long build times, security, and test environments. Their study methodology and context (two industrial case studies) are different from ours (PRs from the open-source). Nevertheless, some of the factors they found are also reflected in the challenges highlighted in our study, for which we also identified mitigation-related actions and changes.

Defects introduced in CPSs can have their own specificity, concerning the presence of CPS-specific bugs. Garcia et al. [20] and Wang et al. [43] studied bugs in two CPS-related domains, autonomous cars and unmanned aerial vehicles, respectively. The presence of CPS-specific bugs, for example, related to an anomalous behavior of a vehicle in the presence of traffic lights may require suitable testing activities, possibly using simulators. This may impact the way CI/CD pipelines are configured.

Concerning CPS testing, Afzal et al. [5], by using interviews with roboticists, identified 12 testing practices, 4 challenges about designing testing platforms tests, and 5 challenges about running and automating tests. While they focused on challenges specific to testing activities, our study focuses on problems faced while setting up a pipeline for CPS development. Defining proper oracles represents a challenge for many CPSs, as the oracle needs to capture the systems' behavior from sensors, or from a camera looking at the system in action, and compare it with the expected one. Jahangirova et al. [25] proposed metrics that capture drivers' behavior that can be used in the context of testing autonomous vehicles. Stocco et al. [38] proposed a deep neural network approach to predict the misbehavior of autonomous cars within a Udacity [4] simulator. The availability of metrics and approaches defined above is functional to the proper setting of CI for CPSs, to achieve testing automation.

Malavolta et al. [29] studied the development of systems based on ROS. For that purpose, they looked at 335 ROS-related repositories, and have built a set of guidelines, validated by robot-development experts. Our work is complementary to them as we focus more on the CI/CD process than on the product. Other related work to ROS development [11, 28] concerns energy consumption of these systems.

6 CONCLUSIONS AND FUTURE WORK

This paper analyzes CI/CD pipelines of 20 open-source Cyber-Physical Systems (CPSs). Specifically, by qualitatively analyzing a sample of 670 pull requests (PRs), we studied challenges and bad practices occurring when setting up and maintaining these pipelines, and their mitigation and restructuring actions.

On the one hand, using a different study protocol, the results confirm several bad practices already known for conventional systems. On the other hand, new challenges (with their mitigation) emerge, and some existing problems are exacerbated in the CPS context. Among others:

Several root causes (and mitigation) for flakiness. Flakiness is a relevant problem in testing conventional systems [18, 27, 35, 47, 49]. For CPSs, the interaction between multiple heterogeneous

hardware and software components, and their (sometimes lost) interconnection through the network worsen the problem. Specific analyses are required to identify flakiness and mitigate/solve it.

Using and integrating simulators is challenging. Simulators are a key asset for CPS development. By looking at the CI/CD pipelines of open-source simulators, of their components, as well as of systems leveraging them, the study shows that simulators undergo evolutionary changes to cope with specific CPS needs. Furthermore, if they are not designed to work within a CI/CD pipeline, their integration becomes more challenging.

Simulators or HiL, this is the question. Simulators may not properly reproduce the HiL behavior or the execution environment in which it operates, and unlikely reflect hardware real-time properties. Sometimes developers decide to use both simulators and HiL in the pipeline, at different stages. This either requires the use of different CI/CD services (a local one and a cloud-hosted one) or local runners for running tasks on HiL. Also, this may require having different builds, with different periodicity, e.g., continuous builds on simulators and periodic builds on HiL.

Complex environment, with multiple devices, operating systems, and firmware versions. This has an impact on setting up the build run-time environment, which set may become difficult to perform and maintain, but also overly slow. Containerization is the preferred solution, and it is achieved by setting up families of predefined images specifically tailored for CPS run-time/simulation needs. Furthermore, the set of parallel jobs/build matrices also require consistent maintenance to (i) keep them up-to-date with devices' evolution, (ii) avoid builds on unnecessary configurations, (iii) keep track of complex dependencies that may limit parallelization, and (iv) use build stages to ensure fast feedback as much as possible.

From the study, it emerges that CI/CD pipelines for CPS development are extremely heterogeneous. Their configuration and evolution vary a lot from system to system. That being said, it would be useful, for future research, to have recommenders automatically suggest pipeline changes every time a kind of problem (or a bad practice) occurs during development. While this has been done in the context of conventional CI/CD pipelines [19, 41], it may be beneficial to conceive recommenders tailored for specific CPS domains. Finally, specific studies on simulator integration patterns and strategies are also desirable.

ACKNOWLEDGEMENTS

This work is supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 957254.

REFERENCES

[1] [n. d.]. Docker security. https://docs.docker.com/engine/security/ Accessed Jan 20 2022.
[2] [n. d.]. Gazebo Robot Simulator. http://gazebosim.org/ Accessed Jan 18 2022.
[3] [n. d.]. PTV VisSim. https://www.ptvgroup.com/en/solutions/products/ptv-vissim/ Accessed Jan 18 2022.
[4] [n. d.]. Udacity.A self-driving car simulator built with Unity. https://github.com/udacity/self-driving-car-sim. Accessed Jan 18 2022.
[5] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. 2020. A Study on Challenges of Testing Robotic Systems. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 96–107. https://doi.org/10.1109/ICST46399.2020.00020
[6] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1199–1210. https://doi.org/10.1109/ICSE.2019.00122
[7] Caio Barbosa, Anderson Uchôa, Daniel Coutinho, Filipe Falcão, Hyago Brito, Guilherme Amaral, Vinicius Soares, Alessandro Garcia, Baldoino Fonseca, Marcio Ribeiro, and Leonardo Sousa. 2020. Revealing the Social Aspects of Design Decay: A Retrospective Study of Pull Requests. In *Proceedings of the 34th Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 364–373. https://doi.org/10.1145/3422392.3422443
[8] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *J. Syst. Softw.* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016
[9] João Brunet, Gail C. Murphy, Ricardo Terra, Jorge Figueiredo, and Dalton Serey. 2014. Do Developers Discuss Design?. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 340–343. https://doi.org/10.1145/2597073.2597115
[10] L. Chen. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software* 32, 2 (2015), 50–54.
[11] Katerina Chinnappan, Ivano Malavolta, Grace A. Lewis, Michel Albonico, and Patricia Lago. 2021. Architectural Tactics for Energy-Aware Robotics Software: A Preliminary Study. In *Software Architecture - 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings*. 164–171. https://doi.org/10.1007/978-3-030-86044-8_11
[12] J Cohen. 1960. A coefficient of agreement for nominal scales. *Educ Psychol Meas.* 20 (1960).
[13] Roland Croft, Yongzheng Xie, Mansooreh Zahedi, Muhammad Ali Babar, and Christoph Treude. 2022. An empirical study of developers' discussions about security challenges of different programming languages. *Empir. Softw. Eng.* 27, 1 (2022), 27. https://doi.org/10.1007/s10664-021-10054-w
[14] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *1st Annual Conference on Robot Learning, CoRL 2017 (Proceedings of Machine Learning Research, Vol. 78)*. PMLR, 1–16.
[15] Santiago Dueñas, Valerio Cosentino, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2018. Perceval: Software project data at your will. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 1–4.
[16] Paul Duvall, Stephen M. Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley.
[17] Paul M. Duvall. 2011. Continuous Delivery: Patterns and Antipatterns in the Software Life Cycle. *DZone refcard #145* (2011). https://dzone.com/refcardz/continuous-delivery-patterns
[18] 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 2019, Tallinn, Estonia, August 26-30, 2019*. 830–840.
[19] Keheliya Gallaba and Shane McIntosh. 2018. Use and misuse of continuous integration features: An empirical study of projects that (mis) use Travis CI. *IEEE Trans. Software Eng.* 46, 1 (2018), 33–50.
[20] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. 385–396.
[21] Georgios Gousios, Margaret-Anne D. Storey, and Alberto Bacchelli. 2016. Work practices and challenges in pull-based development: the contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 285–296. https://doi.org/10.1145/2884781.2884826
[22] Georgios Gousios, Andy Zaidman, Margaret-Anne D. Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 358–368. https://doi.org/10.1109/ICSE.2015.55
[23] Philipp Helle, Wladimir Schamai, and Carsten Strobel. 2016. Testing of Autonomous Systems - Challenges and Current State-of-the-Art. *INCOSE International Symposium* (2016), 571–584.
[24] 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 2017*.
[25] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. 2021. Quality Metrics and Oracles for Autonomous Vehicles Testing. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. 194–204. https://doi.org/10.1109/ICST49551.2021.00030

[26] 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.

[27] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 202:1–202:29. https://doi.org/10.1145/3428270

[28] Ivano Malavolta, Katerina Chinnappan, Stan Swanborn, Grace A. Lewis, and Patricia Lago. 2021. Mining the ROS ecosystem for Green Architectural Tactics in Robotics and an Empirical Evaluation. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. 300–311. https://doi.org/10.1109/MSR52588.2021.00042

[29] 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* (Seoul, South Korea) *(ICSE-SEIP '20)*. ACM, New York, NY, USA, 31–40.

[30] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. 2016. Continuous Integration Applied to Software-Intensive Embedded Systems - Problems and Experiences. In *Product-Focused Software Process Improvement - 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10027)*. 448–457.

[31] Helena Holmstrom Olsson, Hiva Alahyari, and Jan Bosch. 2012. Climbing the "Stairway to Heaven" – A Mulitiple-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 Computer Society, Washington, DC, USA, 392–399.

[32] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why Developers Refactor Source Code: A Mining-based Study. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 29:1–29:30. https://doi.org/10.1145/3408302

[33] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. 2014. Security and Emotion: Sentiment Analysis of Security Discussions on GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 348–351. https://doi.org/10.1145/2597073.2597117

[34] 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* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 164–175.

[35] 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*. 112–122.

[36] P. Soulier, Depeng Li, and J. R. Williams. 2015. A survey of language-based approaches to Cyber-Physical and embedded system development. *Tsinghua Science and Technology* 20, 2 (2015), 130–141. https://doi.org/10.1109/TST.2015.7085626

[37] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[38] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour prediction for autonomous driving systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea*. 359–371.

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

[40] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *ESEC/SIGSOFT FSE*. ACM, 805–816.

[41] 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.

[42] Giovanni Viviani, Calahan Janik-Jones, Michalis Famelis, Xin Xia, and Gail Murphy. 2018. What Design Topics do Developers Discuss?. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 328–3283.

[43] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. 2021. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. 20–31. https://doi.org/10.1145/3468264.3468559

[44] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. 2021. CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 471–482.

[45] Fiorella Zampetti, Vittoria Nardone, and Massimiliano Di Penta. 2022. *Dataset of the manuscript "Problems and Solutions in Applying Continuous Integration and Delivery to 20 Open-Source Cyber-Physical Systems"*. https://doi.org/10.5281/zenodo.5883236

[46] 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.

[47] 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 *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. 50–61.

[48] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Study. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Urbana-Champaign, IL, USA). 12 pages. http://dl.acm.org/citation.cfm?id=3155562.3155575

[49] Celal Ziftci and Diego Cavalcanti. 2020. De-Flake Your Tests : Automatically Locating Root Causes of Flaky Tests in Code At Google. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 736–745.