COSMOS

DevOps for Complex Cyber-physical Systems

Project Number 957254

D3.5 Methodology for setting-up CI/CD pipelines for CPS

Version 1.0
30 June 2023
Final

Public Distribution

University of Sannio

Project Partners: **Aicas, Delft University of Technology, GMV Skysoft, Intelligentia, Q-media, Siemens, Siemens Healthcare, The Open Group, University of Luxembourg, University of Sannio, Unparallel Innovation, Zurich University of Applied Sciences**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the COSMOS Project Partners accept no liability for any error or omission in the same.

© 2023 Copyright in this document remains vested in the COSMOS Project Partners.

Project Partner Contact Information

Aicas James Hunt Emmy-Noether-Strasse 9 76131 Karlsruhe Germany Tel: +49 721 663 968 0 E-mail: jjh@aicas.com	**Delft University of Technology** Annibale Panichella Van Mourik Broekmanweg 6 2628 XE Delft Netherlands Tel: +31 15 27 89306 E-mail: a.panichella@tudelft.nl
Intelligentia Davide De Pasquale Via Del Pomerio 7 82100 Benevento Italy Tel: +39 0824 1774728 E-mail: davide.depasquale@intelligentia.it	**GMV Skysoft** José Neves Alameda dos Oceanos Nº 115 1990-392 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Q-media Petr Novobilsky Pocernicka 272/96 108 00 Prague Czech Republic Tel: +420 296 411 480 E-mail: pno@qma.cz	**Siemens** Birthe Boehm Guenther-Scharowsky-Strasse 1 91058 Erlangen Germany Tel: +49 9131 70 E-mail: birthe.boehm@siemens.com
Siemens Healthineers David Malgiaritta Siemensstrasse 3 91301 Forchheim Germany Tel: +49 9191 180 E-mail: david.malgiaritta@siemens-healthineers.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Sannio Massimiliano Di Penta Palazzo ex Poste, Via Traiano I-82100 Benevento Italy Tel: +39 0824 305536 E-mail: dipenta@unisannio.it	**University of Luxembourg** Domenico Bianculli 29 Avenue J. F. Kennedy L-1855 Luxembourg Luxembourg Tel: +352 46 66 44 5328 E-mail: domenico.bianculli@uni.lu
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**Zurich University of Applied Sciences** Sebastiano Panichella Gertrudstrasse 15 8401 Winterthur Switzerland Tel: +41 58 934 41 56 E-mail: panc@zhaw.ch

Document Control

Version	Status	Date
0.1	Document outline	15 January 2023
0.2	Introduction and Design draft	1 March 2023
0.3	First partial draft	13 June 2023
0.4	First full draft	20 June 2023
0.7	Further editing draft	25 June 2023
0.8	Internal reviews and updates	26 June 2023
1.0	Final QA Version	30 June 2023

Table of Contents

Executive Summary

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 deliverable aims at defining an overall methodology for instantiating, configuring and maintaining CI/CD pipelines for CPS development. Starting by summarizing the main challenges dealing with both the overall CI/CD process in place and the concrete CI/CD pipeline in use within the organization, we will provide a set of guidelines to apply CI/CD in CPS. Specifically, the deliverable provides a tentative CI/CD pipeline that can be used to address some of the challenges experience by developers by properly identifying how and where it is possible to integrate the tools and/or approaches being developed within COSMOS. In doing this, we will specify for each stage the set of tools/approaches that can be integrated together with a brief description of their functionality and highlighting what are the challenges addressable by them. Unsurprisingly, since COSMOS focuses more on trustworthy, self-adaptive CPS, particular attention has been devoted to V&V and security assessment in the pipeline. Tools and approaches leverages various sources, including combine static, dynamic, and historical software analysis techniques, as well as the use of machine learning and deep learning techniques applied on CI/CD and hardware log analysis, to identify CPS-specific issues specific involving hardware-in-the-loop (HiL).

In other words, the methodology being presented places COSMOS approaches and tools around a CI/CD pipeline, i.e., leverages COSMOS techniques and tools to create a CI/CD pipeline for CPS. In doing this, it maps the challenges coming from deliverable D3.2 to the stages mainly included in a CI/CD pipeline for CPS and to COSMOS approaches and tools, where available.

1 Introduction

In recent years we are witnessing a continuous increase in the complexity of the software applications, and the development teams and infrastructure, that are mainly distributed and comprise heterogeneous software and hardware components interacting with each other, i.e., Cyber-Physical Systems (CPSs). As a consequence, the software development and release are becoming very knotty and challenging processes. Furthermore, new development practices are emerging, such as Continuous Integration (CI) and, by extension, Continuous Delivery (CD). The latter implies the use of a fully-automated build and delivery process in order to have multiple integrations and releases per day.

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 [21, 45]. Second, determining a testing scenario for those systems may imply simulating the environment [11] 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 [7, 22, 46]. 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.

Challenges in setting up CI/CD pipelines have also been pointed in previous COSMOS deliverables, and in particular D3.2 [9] and a companion article [49]. While some challenges and bad practices occurring when setting/evolving CI/CD pipelines for conventional systems are exacerbated in the CPSs domain due to their intrinsic complexity there exist also challenges that are specific to CPS development. Specifically, even within a single application domain, a CPS must be able to run/be interfaced with multiple/diverse hardware devices implying “ad-hoc” CI/CD pipeline configurations, e.g., through build matrices for coping with different environments enacted by simulators or HiL. From a different perspective, 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.

To sum up, CI/CD pipelines for CPS development are extremely heterogeneous. Their configuration and evolution vary a lot from system to system. That being said, helping developers in setting up a CI/CD pipeline poses unique challenges.

In this deliverable, we present a methodology for instantiating, configuring and maintaining CI/CD pipelines for CPS development. Starting by summarizing the main challenges dealing with both the overall CI/CD process in place and the concrete CI/CD pipeline in use within the organization, we will provide a set of guidelines to apply CI/CD for CPS development. Specifically, the deliverable provides a tentative CI/CD pipeline that can be used to address some of the challenges experienced by developers by properly identifying how and where it is possible to integrate the tools and/or approaches being developed within COSMOS. In doing this, we will specify for each stage, the set of tools/approaches that can be integrated together with a brief descrip-

tion of their functionality and highlighting what are the challenges addressable by them. Unsurprisingly, since COSMOS focuses more on trustworthy, self-adaptive CPS, particular attention has been devoted to V&V and security assessment in the pipeline. Tools and approaches leverages various sources, including combine static, dynamic, and historical software analysis techniques, as well as the use of machine learning and deep learning techniques applied on CI/CD and hardware log analysis, to identify CPS-specific issues specific involving hardware-in-the-loop (HiL). In other words, the methodology being presented places COSMOS approaches and tools around a CI/CD pipeline, i.e., leverages COSMOS techniques and tools to create a CI/CD pipeline for CPS. In doing this, it maps the challenges coming from deliverable D3.2 [9] to the stages mainly included in a CI/CD pipeline for CPS and to COSMOS approaches and tools, where available.

2 Background

This section provides information on (i) CI/CD process, (ii) CPS development, and (iii) CI/CD good and bad practices, focusing more on the challenges faced in CPS development and in CI/CD.

2.1 CI/CD process

Continuous Integration (CI) is a modern development practice in which developers integrate their changes into a shared repository as often as possible. Each change is validated by creating a build and running automated tests and code analysis tools. This practice gives a great emphasis to testing automation aimed at guaranteeing that the software product is not broken when new changes are integrated with the main branch. In this way, it is possible to avoid the integration hell that takes place when the integration is demanded very-closest to the release day.

Continuous Delivery (CD) can be seen as a development practice that embraces CI. CD adoption allows releasing new changes to the customer quickly and in a tenable way. In other words, the release process is completely automated giving the possibility to release daily, weekly, or whenever appropriate (i.e., business requirements are met). Obviously, it is required that the deployment to production occurs as early as possible making sure that the release takes place in small batches.

Continuous Deployment goes one step further than CD. In this practice the software release to the customers occurs without any human intervention. In this way, it is possible to quicken the feedback loop with your customers. In other words, developers can focus on building software, and they see their work go live minutes after they have finished working on it.

Based on the above definitions, it is possible to state that CI is part of both CD and continuous deployment, while continuous deployment is like CD, except that releases happen automatically.

Obviously, there is a cost to be paid for implementing each development practice, but this is overbalanced by their gain. Among the benefits, we mention (i) fewer bugs are brought into production dues to continuous testing, (ii) the overall release process is simplified as integration errors are discovered as soon as they are introduced, (iii) frequently releases can speed up the feedback loop with the customers, and (iv) continuous stream of improvements are perceived by the customers guaranteeing a continuous quality enhancement.

To summarize, the result of applying CI/CD pipeline is that there is always a stable piece of software that works properly and contains few bugs, reducing the time spent in trying to find bugs since they will show up very quickly.

Among the principle ingredients needed for applying CI and CD development practices in a suitable manner there are (i) “automation” implying that anyone submitting a single command can activate the whole process and (ii) the adoption of a well-defined build strategy that, for instance, based on the type or size of the change is able to execute the whole set of build tasks, or only a subset (i.e., it is not a good practice to run time and/or resource-consuming tasks at each change). From a different perspective, adopting CI/CD processes allows

improving communication among teams that are, quite often, spread around the world and are increasing in size. The improvement in communication is strictly related to the "rapid feedback" rule meaning that developers have to be notified as soon as possible of problems introduced in the main branch in order to fix the builds breakages very quickly, guaranteeing to have always a stable version of the software product. The latter has to be verified in a production-like environment implying the need for configuring at least one test environment that is as similar as possible to the production one. Obviously, there are cases in which the setting of an environment very similar to the production one is prohibitive. For this reason, it is required to analyze the risks related to any kind of difference between the test and the production environment.

As regards the continuous deployment, there is the need for having scripts that could be executed automatically guaranteeing an automated deployment (i.e., improve speed and reduce errors) that has to account for automated roll-back strategies in order to allow an automatic revert in presence of failures or errors on the mainline or in production making sure that everything is in good shape.

In recent years, researcher have studied the CI/CD practices adopted in industry and open source since that, as also confirmed by Hilton et al. [23], CD, and consequently CI, is becoming increasing popular. As regards the adoption of CI in industry, Laukkanen et al. [29] observed that developers face many technical and social challenges when adopting CI, such as the test infrastructure. Moreover, Ståhl and Bosch [43], starting from a literature review and conducting an in-field study have found that different software development projects use different CI implementations because of several contextual factors such as longevity, size, budget, competences, organizational structure, or geographical distribution. This implies that in industry there is not a uniform adoption of CI. The latter means that there are many variation points in the usage of the CI term. From a different perspective, Elazhary et al. [15] looked at the extent to which companies follow the CI practices by Fowler and Foemmel [17] through interviews. Their results emphasized differences among companies in terms of repository structure, testing automation, long build, and deployment challenges. A recent study by Vassallo et al. [47] found that not only CI practice varies between different industrial companies, but there are noticeable differences between OSS and industry discovered observing the type of build failures.

Going more deeply on the CD practices, instead, Rahman et al. [38] have reported on a study aimed to understand the software practices used in CD in 19 software companies, e.g., Facebook, Netflix and Rally Software. Their results highlight that, in applying CD, the software companies use automated testing and automated deployment, i.e., the two pre-requisites for continuous deployment. CD also necessitates the consistent use of other software practices such as code review, monitoring and shepherding changes. The latter supports the basic principle of DevOps since that developers are getting more involved in operations not only on development.

It is important to understand the relation between the usage of CI/CD pipeline and both code quality and developers' productivity. As an example, Miller et al. [33] reported that in Microsoft the CI cost was about 40% of the cost of an alternative solution (without CI) process achieving the same level of quality. Moreover, Deshpande and Riehle [10], analyzing commit data from open source projects, observed that in OSS the adoption of CI has not influenced development and integration practices. From a different perspective, Vasilescu et al. [46] mined GitHub projects and found that CI makes teams more productive and improves the likelihood of pull request merges, without sacrificing the overall project quality.

Looking more deeply on the benefits related to the adoption of CD practices, Savor et al. [40] reported on an empirical study done in two industrial (Internet) companies with the goal of investigating how Continuous Deployment is used in industry. Their results highlighted that the CD adoption does not limit the scalability in terms of productivity of one organization even if the system grows in size and complexity.

A key point that has to be investigated when there is the adoption of a new practice is the one aimed to understand the impact that this practice has on build maintenance activities in terms of maintenance effort. Specifically, McIntosh et al. [32] noticed that, in OSS the effort involved in maintaining the build configuration can introduce 27% overhead on source code development and a 44% overhead on test development.

2.2 Development of Cyber-Physical Systems

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 are more complex and difficult to design, develop, test, and integrate than conventional software systems [21, 30, 44, 45]. Specifically, Törngren et al. [45] 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 [20], as well as testing encapsulating model-in-the-loop (MiL) [42], software-in-the-loop (SiL), and hardware-in-the-loop (HiL) [2]. Differently from traditional software, when developing CPS, conformance with requirements is often verified through testing executed at different development stages, based on available development artifacts [39]. Depending on the development stage, testing of CPS software is performed through simulators or with hardware in the loop. On the one hand, simulation usually occurs in the model-in-the-loop (MiL) and software-in-the-loop (SiL) stages where the physical systems are simulated. On the other hand, hardware is used both when the system is validated on the real production environment and in the HiL where the software components of the CPS are deployed on the final hardware while the environment is simulated.

Considering the costs, risks, and complexity of conducting system testing in a real environment [11, 28], simulation is becoming one of the cornerstones in developing and validating CPSs. The usage of CPS simulation environments enables automated test generation and execution [25, 34]. However, the limited budget allocated for testing activities and the virtually infinite testing space pose challenges for adequately exercising the CPS behavior [3, 16, 48].

Related to DevOps applications in a CPS context, Park et al. [36] 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 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.

Finally, Mårtensson et al. [31] 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 who want to adopt CI for embedded systems.

2.3 CI/CD adoption challenges and bad practices

Hilton et al. [22] have investigated the barriers that developers face in moving towards CI in both industrial and open source projects. They identified trade-offs involving different and orthogonal dimensions namely: (i) assurance, i.e., the usage of a CI pipeline has to proof that the code is correctly tested, keeping the overall build times manageable, (ii) security, i.e., to ensure the integrity of the code and to protect sensitive information that is needed in the execution of the build process but also machines that are used to run the CI system, and (iii) flexibility, i.e., to have a great amount of configuration options that results in having a very powerful CI system. However, very often the latter implies increasing the complexity of the CI system that is something that developers do not desire. Moreover, it is well known that highly configurable systems hinder their usability.

Focusing more on the CD practices, Chen [8] analyzed four years' CD adoption in a multi-billion-euro company and elaborated a list of challenges related to CD adoption. Furthermore, Chen identified six strategies that can be used in order to overcome the aforementioned challenges such as (i) selling CD as a painkiller, (ii) starting the adoption of CD with easy but relevant applications/projects, and (iii) visualizing the CD pipeline skeleton. From a different perspective, Olsson et al. [35] explored the barriers faced by developers in moving towards CD based on the evolution path in 4 different software development companies such as the lack

of a base product on which developers can do continuous improvements, the lack of a well-defined process used for collaborating and exchanging information between different teams, but also the usage of old tools that make the work unnecessarily difficult. Going more in-depth on the barriers encountered when moving from CI to CD, Olsson et al. [35] identified as the most relevant barrier the one related to the complexity of the environments (in particular network environments) where the product should be deployed.

Zhang et al. [50], instead, focused on a specific (r)evolution of the CD pipeline namely: the *containerization* with Docker. More specifically, by means of a survey they identified the motivations, usage (i.e., CD workflows), needs and barriers when adopting containerized CD. After that, Zhang et al. tested the main findings coming from the survey by means of a large-scale quantitative study. Their results shed light on the benefits related to the usage of CD pipeline like the replacement of manual deploy that implies the possibility to have a production reliable (i.e., less error prone), the possibility to catch errors earlier that minimizes the number of failures, but also the optimization in terms of time spent on development (i.e., since the configuration is not treated like code it is unavoidably reduced the time spent on maintenance or configuration activities).

Once CD is in place, it might be wrongly applied, thus limiting its effectiveness. Indeed, previous work [12, 13, 24] highlights some bad practices in its exercise concerning commits frequency, management of build artifacts, and overall build duration. Such bad practices highlighted the need for (i) a fully automated build process, (ii) a centralized dependencies management in order to reduce class-path and transitive dependencies' problem, (iii) run private builds, and (iv) the existence of different target environments on which deploy candidate releases. Humble and Farley [24] set out the principles of software delivery and provided suggestions on how to construct a delivery pipeline, by using proper tools, automating deployment and several testing activities.

In this context, Duvall [14], by surveying related work in literature, created a comprehensive catalog featuring 50 different patterns and antipatterns regarding several phases or relevant topics in the CD process. In the catalog [14], it is possible to find bad habits concerning the versioned resources, the way developers use to trigger a new build, test scheduling policies, and the use of feature branches. Duvall also highlighted the need for a fully automated CD pipeline having a proper dependency management, a roll-back release strategy and production environments where the release candidate should be tested.

Gallaba and McIntosh [19] conducted an empirical study aimed at investigating which CI features provided by Travis CI are being (mis)used showing that $\simeq 48\%$ of the configuration files are related to build job processing nodes meaning that developers rarely use the CI service provider for implementing continuous deployment (2%). Moreover, very often the CI specifications once committed rarely change.

DevOps organizations use specific technologies to automate repetitive work. In particular, it is common practice in CI/CD to treat configurations and infrastructure specifics of the deployment environments as code in form of scripts, i.e., infrastructure as code (IAC) scripts. Similarly to production and test code, also IaC scripts change frequently during the software evolution and these changes can unavoidably introduce defects. Based on the above practices and assumptions, Sharma et al. [41] proposed a catalog accounting for 13 implementation and 11 design configuration smells. Each smell has been defined taking into account the violations to recommended best practices for configuration code. The presence of configuration smells could hinder the maintainability of the overall configuration code. Their results highlight that design configuration smells have more chance for co-occurring compared to the implementation ones. Consequently, a sub-optimal design decision may impact negatively the quality of the configuration code, implying that developers have to take design decisions very diligently. The most surprising result, instead, is the one reporting that increasing the size of a project introduces a decrease in the design configuration smells density. In other words, the latter result contradicts the common wisdom highlighting that the complexity (and the smell density) is highly correlated with the size of the project.

In a more recent work, Rahman and Williams [37] have determined three properties that characterize defective IaC scripts. More specifically, their results highlight that defective IaC scripts are characterized by the usage of file-related operations (i.e., erroneous file mode and file path) that require the setting of specific values like file location and permissions. Moreover, infrastructure provisioning requires the execution of complex steps, e.g., installation of third-party components, that can introduce defects as also the management of a large number of

user accounts. The aforementioned study is based on the findings reported in a study by Jiang and Adams [26] showing that IaC files are quite large and change relatively frequently. Most important, they find that the changes to IaC files are tightly coupled to changes to the test files, as a consequence of the addition of new test cases or the modification of the test case configuration.

3 CI/CD pipeline for CPS

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 traditional software [21, 30, 44, 45].

In such a context, adopting effective Continuous Integration and Delivery (CI/CD) practices off the DevOps menu is extremely relevant and appropriate for setting the execution environment, e.g., Hardware-in-the-Loop (HiL) or simulators. However, existing CI/CD technology cannot be applied to CPSs as is [27]. As an example, 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, would make the application of CI/CD challenging at best.

This section details the steps mainly involved in a CI/CD pipeline for CPSs and how their configurations differ from traditional software, and maps process- and pipeline-related challenges (main outcome of D3.2) on them. Specifically, in determining the mapping we have carefully read what interviewees reported when describing the challenges trying to identify what are the steps in which researchers need to push more effort to provide useful approaches and tools aimed at mitigating them.

3.1 Steps involved

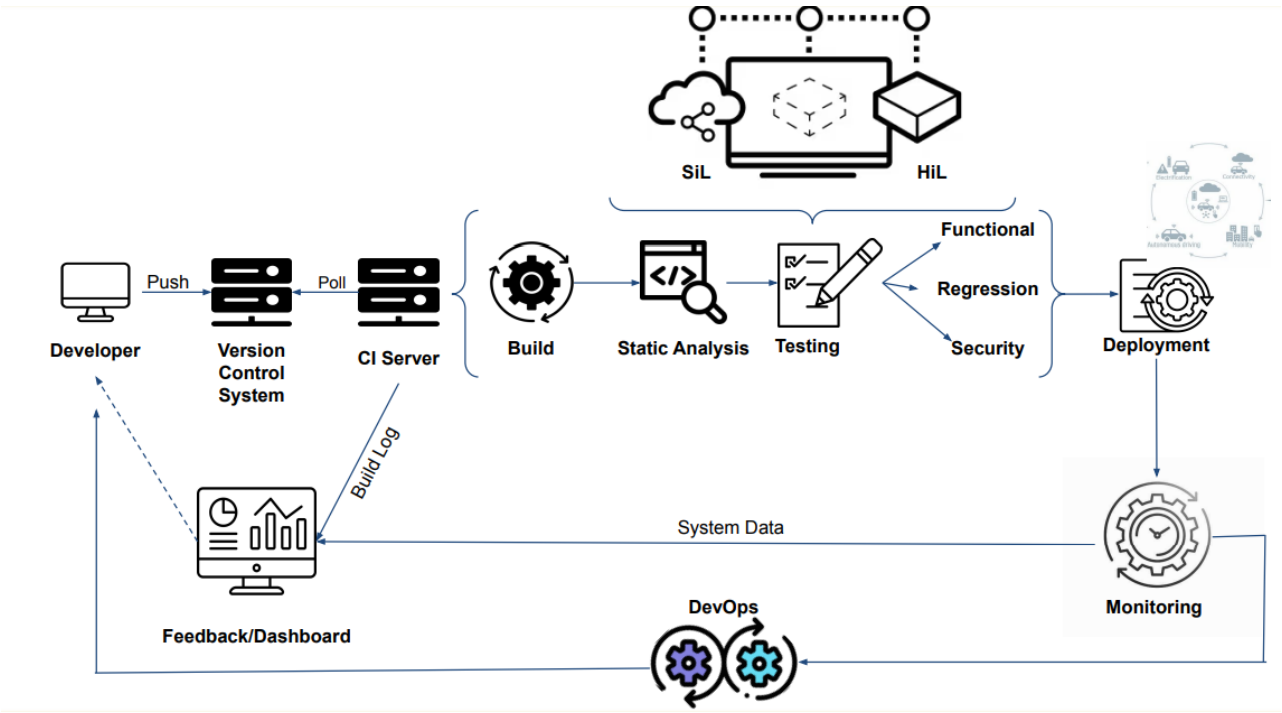

Figure 1: CI/CD pipeline for CPSs

The main goal of a CI/CD pipeline is to reduce the risks involved in deploying software by providing a quick feedback loop on the quality of the software, and by making the deployment process easily-repeatable. Figure 1 depicts the main ingredients to consider when trying to setting up a CI/CD pipeline for developing CPSs. It is important to highlight that, there is no silver bullet for structuring CI/CD pipelines across different CPS domains. Specifically, the "no silver bullet" concept outlined by Brooks [6] is emphasized when it comes to CPS development; even within a single domain, a given program must be able to run/be interfaced with multiple/diverse hardware devices. Thus, the pipeline often needs ad-hoc configuration, e.g., through build matrices for coping with different environments enacted by simulators or HiL. Furthermore, very often CPS development concerns safety-critical systems, and this implies following certain SIL levels [5, 18], such as a pipeline may belong to a validation organization that may have limited/no access to hardware, simulators, or even to the production code, e.g., the development and the V&V teams and pipelines must be kept distinct.

As with traditional software, the CI/CD pipeline is enacted each time a developer wants to push her changes to the stable release branch of the system under development. In other words, the pipeline is triggered automatically when new code is committed to the repository, i.e., the CI server continuously poll the versioning control system in order to identify new changes to be verified before the integration. In the initial step, the CI server will check out the code from the source code repository, e.g., GitHub or GitLab taking into account the specific commit that has triggered the pipeline. At this point, it is possible to start the process by building the code, e.g., if you are developing in a compiled language like Java, the first thing to do is compile the program. If the building process ends successfully, it is possible to move to the next step, i.e., statically analyzing the code to (i) verify the compliance with coding standards, (ii) spot maintainability issues, and (iii) identify software bugs as soon as they are introduced. If the previous step ends with a green status, it is possible to start the testing phase in order to verify that all functional and non-functional e.g., performance or security, requirements are met. The testing phase can be done involving different kind of environments at different development stages, software-in-the-loop or hardware-in-the-loop. Once the testing phase ends successfully, it is possible to deploy the system in production. Once executed all the steps defined in the CI/CD process or else, as soon as one of the steps ends in a failing status, a detailed report is sent as notification to the development teams. Furthermore, once the system is in production it is possible to start the monitoring phase where data collected from the field are sent back to the developers.

Going deeper on the static analysis phase, very often when developing CPSs the software has to be certified, i.e., must meet standards dictated by the application domain. The latter constrains the configuration of the static code analysis tools (SCATs) used within the CI/CD pipeline. Due to the safety integrity level of the CPS, the software must be certified, e.g., the software must be developed following the Motor Industry Software Reliability Association (MISRA) standards [1, 4]. In this context, both development and V&V teams must properly configure SCATs such as SonarQube for (i) checking the fulfillment of the MISRA rules for certification, (ii) identifying maintainability problems such as the presence of duplicated code, and (iii) spotting bugs as soon as they are introduced.

When developing CPSs, conformance with requirements is often verified through testing executed at different development stages, based on available development artifacts [39]. Specifically, in the early development stages, testing of CPS software is performed through simulators, i.e., the physical system or the environment is typically simulated. After that, developers might rely on HiL and only on the last mile, when the version of the software is considered stable, the system is deployed on a real environment and the testing phase is conducted by exercising the real system in a production-like environment. Of course, there is not a strict rule to be followed on deciding which kind of environment to include in the development process. Specifically, on the one hand, when the environment is too complex to be simulated and do not exist domain-specific simulators being accessible, development teams might decide to skip the software-in-the-loop testing while relying directly on the HiL testing. On the other hand, when the cost of the hardware components is too high to be afforded by a company, development teams might decide to do most of the testing activities only relying on simulated environments, so skipping the HiL testing. In this case, if everything is green when exercising the system in a simulated environment, it is possible to deploy on the real system and conduct the testing of the software in a production-like environment.

Table 1: Set of process-related challenges that can be mitigated by properly integrating approaches/tools within the CI/CD pipeline

Challenge	Dev.	Static Analysis	Testing	Digital Twin	Deploy	Continuous Monitoring	DevOps
Cycle-time reduction		✓	✓	✓	✓	✓	✓
Complexity of the env.				✓			✓
Variability of the env.	✓						
Lack of redundancy in the env.	✓						
Tests manually derived			✓				
Tests manually executed			✓	✓			✓
Different interpretation for the same requirements	✓		✓				
Complexity in oracle specification for test automation			✓				✓
Complexity for deriving integration tests			✓				
Complexity for deriving safety tests			✓				
Complexity for oracle automation with simulators			✓				✓

A different aspect that must be considered when deciding the environment to use for testing purposes is the type of requirements under test. In other words, it is possible that specific testing activities cannot be done relying on simulators so must be done by directly including the physical components in the loop.

Finally, another aspect to consider is the overall build execution time that might vary based on whether or not the software need to be deployed on the hardware before being tested. Specifically, developers might prefer to rely on simulated environments to obtain fast feedback in terms of impact of their changes on the overall system during normal working-hours while relying on HiL during nightly builds so that the day after they will have an overall view of how the system behaves based on the last changes.

3.2 Process-related challenges impacting the CI/CD pipeline steps

The interview-based study conducted within D3.2 revealed 20 process-related challenges grouped into six different categories: general, culture, environment, testing, deployment and simulators. Since that those challenges deal with the CPS development process being adopted within the organizations, it is possible that some of them cannot be mitigated by relying on approaches/tools being involved in the CI/CD pipeline. On the one hand, the developers' on-boarding (PRC_2) or the limited CI/CD culture for CPS development (PRC_4) cannot be mitigated/overcame by the integration of external tools/approaches into the adopted CI/CD pipeline. On the other hand, it might possible to mitigate the practice of manually deriving/executing test cases (PRC_{10}, PRC_{11}) or else the complexity in oracle specification for test automation (PRC_{14}) by relying on dedicated approaches/tools to be integrated in the CI/CD pipeline. Specifically, Table 1 reports the 11 process-related challenges that might be mitigated by properly integrating approaches and tools within the CI/CD pipeline. Furthermore, for each challenge it is possible to identify what are the phases mainly affected by them. Note that the phases are the same being highlighted in Figure 1.

Adopting a CI/CD development process has as main advantage the overall cycle time reduction, i.e., reducing the feedback loop between the new implemented features and the overall customers' satisfaction. To reduce the cycle time it is important to improve the overall time needed to execute the whole CI/CD phases on the available environments. In doing this it might be required to have approaches and tools able to automatize and speed up all the phases needed to check the conformance of the system against requirements. For instance, it is important to improve the testing phase to spot bugs as soon as they are introduced, as well as the deployment phase where it is important to provide tools able to help developers in properly locate deployment-related errors. Also for the static analysis phase it might be of interest to include tools able to identify performance leakage.

Looking at the process-related challenges dealing with the environment, the ones that might be mitigated by properly setting a CI/CD pipeline deal with the characteristics of the physical environment in which the developed code has to be deployed. Specifically, the complexity of the environment has an impact on whether developers might rely on simulators or have to do almost all of the work directly on the hardware components.

Table 2: Set of pipeline-related challenges dealing with the overall pipeline properties that can be mitigated by integrating tools within the CI/CD pipeline

Challenge	Dev.	Static Analysis	Testing	Digital Twin	Deploy	Continuous Monitoring	DevOps
Long build execution time			✓	✓	✓		✓
Build time estimation						✓	
SCATs configuration		✓					✓
Lack to access production code from the pipeline	✓		✓				
CI/CD config. coupled with the env.							✓
Re-usability of build artifacts							✓

The main issues is related to the availability of simulators that in such complex environments are complex to self-develop within the organization. Moving onto the variability of the environment, there are application domains where it is mandatory to verify the conformance to both functional and non-functional requirements against multiple versions of the hardware devices. In these domains it is important to have proper tools helping developers in customizing the developed code so that it will take into account the high variability of the physical components. Similar tools are needed to mitigate/overcome the last challenge dealing with the structure of the adopted development process that has no redundancy implying that in presence of network or server issues it is not possible to access the code under development.

Moving onto the testing phase of the CI/CD pipeline, six out of seven challenges might be mitigated by including appropriate tools within the CI/CD pipeline, but also providing tools helping developers in properly specify test cases specifically for what concerns the integration testing, i.e., complexity to determine the expected behaviour of the system on the whole, and the non-functional testing, e.g., security. As regards the latter, the main difficulty has to deal with the identification of situations that could never happen or that are not expected to happen. Finally, when specifying test cases, one challenge deals with the complexity in specify an automated oracle impacting the overall CI/CD setting in terms of phases being automatized.

The last challenge in Table 1 deals with the usage of simulators in the overall development process. Specifically, it is related to the complexity faced when automatizing the test execution due to the impossibility to properly specify an oracle. In other words, if it is complex for a human to specify the expected behaviour for some scenarios, it is very unlikely to have simulators that can properly emulate that behaviour. Similar to previous cases, it is important to have tools and approaches able to properly determine the oracle by for instance automatically learning what the expected behaviour of the system would be in specific scenarios.

3.3 Pipeline-related challenges impacting the CI/CD pipeline steps

The interview-based study conducted within D3.2 revealed 29 pipeline-related challenges properly grouped into five categories reflecting specific aspects of the CI/CD pipeline setting and evolution: pipeline properties, thoroughness, simulators, HiL, and flaky behaviour. Differently from the process-related challenges, all the pipeline related challenges might be mitigated by properly tools and approaches helping developers to deal with them. In the following we will provide a detailed description of how the challenges impact the different elements in a CI/CD pipeline for CPSs shown in Figure 1.

3.3.1 Pipeline Properties

Table 2 reports the six pipeline-related challenges dealing with the build execution time and the overall CI/CD pipeline configuration that might be mitigated by properly integrating approaches and tools within the pipeline. For each challenge it is possible to identify what are the phases (as shown in Figure 1) mainly affected by them.

The build execution time influences the set of phases automatized within the pipeline, and differently from conventional systems, when developing CPSs the challenge is further exacerbated upon deploying and executing the software on simulators and/or HiL. One possibility to reduce the overall build execution time is to integrate

Table 3: Set of pipeline-related challenges dealing with thoroughness that can be mitigated by integrating tools within the CI/CD pipeline

Challenge	Dev.	Static Analysis	Testing	Digital Twin	Deploy	Continuous Monitoring	DevOps
Development env. detached from execution env.			✓				
Difficulty for detecting deployment related errors					✓		✓
Continuous installation					✓		✓
Closing the loop causing performance degradation				✓		✓	✓
Complexity due to uncontrollable factors				✓		✓	✓
Complexity due to data from the field				✓		✓	✓

tools able to prioritize test cases so that the ones having the highest failing probability are executed at the beginning, or else select only a subset of the test cases to be executed by taking into account the type of change triggering the CI/CD process and the affected components. The difficulty to perform build time estimation, i.e., the amount of time required for the overall build execution, instead, is related to the used infrastructure so it is highly dependent on external factors like the time needed to acquire and elaborate huge amount of data from the field, i.e., monitoring stage.

Moving onto the challenges related to the pipeline configuration, for what concerns SCATs, very often there are no clear coding standards or guidelines so it would be helpful to have approaches able to apply coding style inference.

Differently from traditional software, the safety integrity level of the CPS under development might require the separation between the development and the V&V teams with the latter having no access to the production code, *"need to guarantee the protection of the source code"*. Furthermore, both technology and deployment infrastructure might limit the appropriate setting of the overall pipeline. In these cases it is desirable to have tools able to helps developers to properly set and customize the CI/CD pipeline likely relying on knowledge acquired by pipeline set for similar systems relying on pretty similar technologies and infrastructures. Finally the impossibility to re-use previously built artifacts constraint the overall build execution time that will unavoidably increase.

3.3.2 Pipeline Thoroughness

Table 3 reports the six pipeline-related challenges experienced when (i) ensuring the effectiveness of the pipeline, i.e., first three challenges, and (ii) closing the DevOps loop by gathering data from the real physical environment, i.e., last three challenges, that might be mitigated by properly integrating approaches and tools within the pipeline. For each challenge it is possible to identify what are the phases (as shown in Figure 1) mainly affected by them.

The challenges dealing the overall accuracy and completeness of the CI/CD pipeline impact the tasks automatized within the pipeline. First of all it is important to provide developers with tools able to properly adapt the pipeline configuration automatically based on the changes to the physical environment, as well as tools able to locate and isolate deployment-related errors for development processes relying on incremental deployments.

As regards the need to closing the DevOps loop, the three challenges are all related to complexities in acquiring data from the field. For mitigating these challenges it is desirable to provide developers with tools able to control the performance of the monitoring step that could become invasive with respect to the overall system performance. Furthermore, very often there are uncontrollable factors in real execution environments so developers need approaches helping them in properly identifying the presence of noise from data acquired from the field that can also help them in guaranteeing the quality of data collected from the field used to make informed decision about whether or not the system behaves as expected.

Table 4: Set of pipeline-related challenges dealing with simulators that can be mitigated by integrating tools within the CI/CD pipeline

Challenge	Dev.	Static Analysis	Testing	Digital Twin	Deploy	Continuous Monitoring	DevOps
Limited functionality	✓			✓			✓
Functional correctness	✓			✓			
Real-time properties			✓	✓			
Interaction with env.	✓			✓			✓
Accessibility				✓			✓

Table 5: Set of pipeline-related challenges dealing with HiL that can be mitigated by integrating tools within the CI/CD pipeline

Challenge	Dev.	Static Analysis	Testing	Digital Twin	Deploy	Continuous Monitoring	DevOps
Availability				✓	✓		✓
Automated deployment on HiL					✓		
Test automation on HiL			✓	✓			✓
Costs and Scalability			✓	✓			✓
HiL standalone			✓	✓			

3.3.3 Simulators

Table 4 reports the five pipeline-related challenges due to issues and limitations of simulators used for the software-in-the-loop stage. For each challenge it is possible to identify what are the phases (as shown in Figure 1) mainly affected by them. All those challenges have an impact on the choice of the execution environment, i.e., relying on simulators or HiL for verification and validation purposes within the pipeline. Almost all the challenges originate from the high complex physical environments that must be simulated. For addressing them it would be desirable to have tools able to automatically learn the behaviour of the physical systems and transform this knowledge in features of the simulators improving the overall trustworthiness.

3.3.4 HiL

Table 5 reports the five pipeline-related challenges due to issues and limitations of adopting hardware-in-the-loop within the pipeline. For each challenge it is possible to identify what are the phases (as shown in Figure 1) mainly affected by them.

As regards testing on HiL, the complexities are mainly due to limited human resources being available or else the high cost and lack of scalability of the hardware devices/systems. These have an impact on the type of execution environment being used within the pipeline and, unfortunately cannot be mitigated by simply developing tools and approaches to be integrated within the pipeline. It might be possible to do almost all the work for verification and validation purposes relying on simulators that are almost as much similar as possible to the real physical components. The same applies to deployment on HiL.

As regards instead the need to check the availability of the hardware devices before starting the testing phase with the hardware-in-the-loop to reduce the feedback time as well as the time needed for checking the reasons behind a build failure, it is needed to properly configure the CI/CD pipeline so that developers are notified as soon as possible about the impossibility to start the testing phase due to the hardware components being unavailable. The latter can also be used to reduce the flakiness introduced in the process as a consequence of the impossibility to properly communicate with the physical components.

Table 6: Set of pipeline-related challenges dealing with flaky behaviour that can be mitigated by integrating tools within the CI/CD pipeline

Challenge	Dev.	Static Analysis	Testing	Digital Twin	Deploy	Continuous Monitoring	DevOps
Dependencies installation	✓						✓
Features' interaction			✓	✓			
HiL (un)availability				✓			
HiL inputs			✓	✓	✓	✓	✓
Lack of control over hardware resources			✓	✓			
Network issues			✓	✓			
Timing issues			✓	✓			

3.3.5 Flaky behaviour

Table 6 reports the seven root causes that may lead to non-determinism in the build execution used for CPS development. While some of them are pretty similar to the ones experienced also when developing conventional software, there are some root causes that are specific to CPSs. Going deeper on the latter, there could be root causes dealing with the usage of simulators that are not able to cope with timing issues or that are not able to control all the possible interaction between different features mainly related to the complexity of the systems to be simulated. As regards instead the HiL it is possible to have non-determinism related to the presence of noise in the data acquired from the field, as well as due to the unavailability of the hardware leading to a build process failing intermittently. In these cases it is required to develop CPS-specific flaky behaviour detection tools.

4 COSMOS approaches and tools around a CI/CD pipeline for CPS

Figure 2 shows how the tools and approaches developed within COSMOS could be mapped onto the stages of a general CI/CD pipeline for CPS development. Specifically, there are some tools/approaches that can work locally and help developers in properly setting the CI/CD pipeline based on different factors such as the type of change or else the amount of limited resources being available, e.g., FAT CONTAINER DETECTOR or WORKFLOW COMPLETION. Furthermore, there are tools and approaches mainly on specific phases mainly included in a CI/CD process such as static analysis and testing. The latter is mainly differentiated into three different type of activities, namely functional, regression and security. As regards the testing phase, it is important to highlight that when developing CPSs developers make an intensive usage of simulation-based testing before testing directly on the hardware devices, i.e., HiL testing. Each type of testing comes with its own challenges. Within COSMOS, it has been possible to develop tools able to help developers overcoming the challenges experienced in both of them. While TEA, SDC-SCISSOR, and TEASER mainly work within simulation-based testing, MORLOT and SURREALIST can be used both for software- and hardware-in-the-loop testing. Finally, as regards the set of tools and approaches mapped onto the feedback/dashboard component, it is important to state that those are the tools that provide useful information to developers that leveraging them could improve their decision-making process.

Table 7, instead, summarizes the set of both process- and pipeline-related challenges that can be (partially) addressed integrating the tools and approaches being developed within COSMOS. Unsurprisingly, since that during CPS development the most critical phase is represented by V&V, and COSMOS focuses more on that aspect, many of the challenges that could be overcame are the ones dealing with the testing. More in detail, most of the tools and approaches aim at helping developers with the automatic generation of simulation-based test cases, as well as with the automatic definition of "challenging" test input but also with the oracle specification problem.

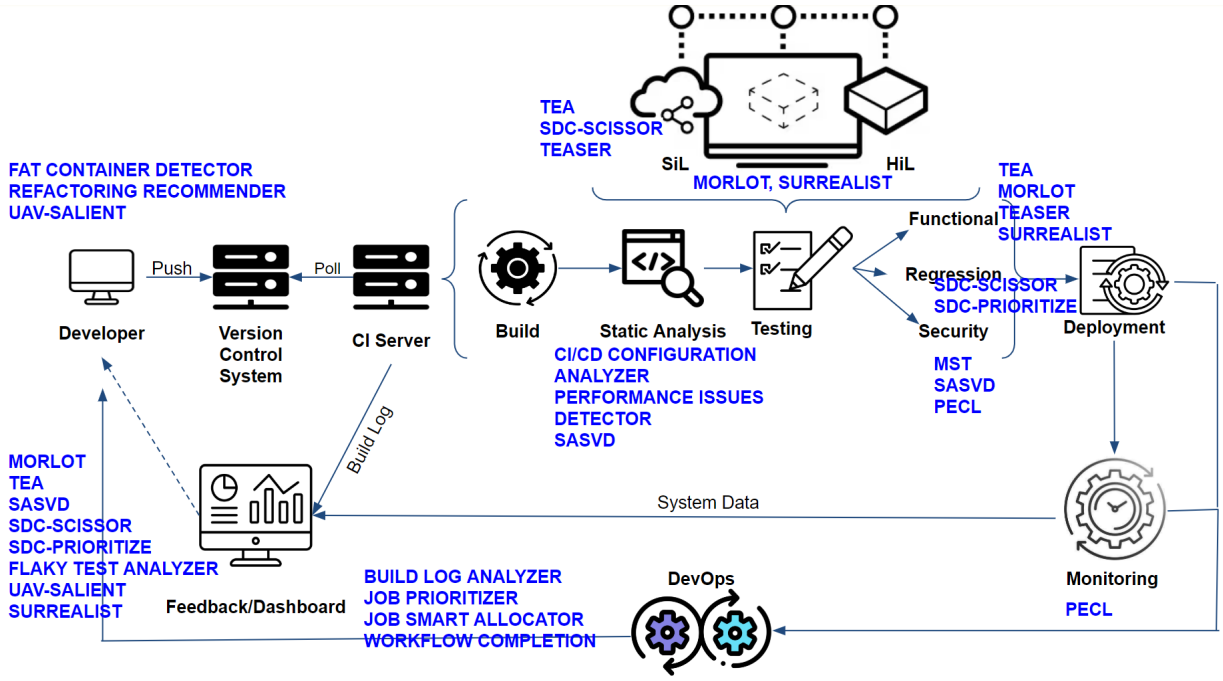

Figure 2: Mapping of the COSMOS Tools onto a general CI/CD pipeline for CPS development

Table 7: Challenges (partially) addressable by tools and approaches developed within COSMOS

Challenge	Tools and Approaches
Cycle-time reduction	FAT-CONTAINER DETECTOR, BUILD LOG ANALYZER, JOB PRIORITIZER, JOB SMART ALLOCATOR, SDC-PRIORITIZE
Complexity of the env.	TEA, MORLOT, SDC-SCISSOR, TEASER, SURREALIST
Variability of the env.	TEA, MORLOT, SDC-SCISSOR, TEASER, SURREALIST
Tests manually derived	TEA, SDC-SCISSOR, TEASER, MORLOT, SURREALIST, MST, SASVD
Tests manually executed	TEA, SDC-SCISSOR, TEASER, MORLOT, SURREALIST, MST, SASVD
Complexity in oracle specification for test automation	PECL, MST
Complexity for deriving safety tests	PECL, MST
Long build execution time	FAT-CONTAINER DETECTOR, BUILD LOG ANALYZER, JOB PRIORITIZER, JOB SMART ALLOCATOR
CI/CD config. coupled with the env.	WORKFLOW COMPLETION
Difficulty for detecting deployment related errors	FLAKY TEST ANALYZER
Functional correctness	TEA, SDC-SCISSOR, TEASER
Real-time properties	TEA, SDC-SCISSOR, TEASER
Interaction with env.	TEA, SDC-SCISSOR, TEASER
Accessibility	FLAKY TEST ANALYZER
Test automation on HiL	MORLOT, SURREALIST
Costs and Scalability	MORLOT, SURREALIST
Dependencies installation	FAT CONTAINER DETECTOR, FLAKY TEST ANALYZER
Features' interaction	FLAKY TEST ANALYZER
HiL inputs	FLAKY TEST ANALYZER
Timing issues	FLAKY TEST ANALYZER

The following subsections will provide a detailed description about how the tools and approaches being developed within COSMOS could be integrated within a CI/CD pipeline for CPS development, together with a detailed description of the challenges that could be (partially) addressed relying on them.

4.1 CPS development

The development of CPSs is a complex task due to the intertwine of various hardware and software components. In this context, bugs or unplanned behaviour of the system can lead to detrimental consequences and accidents involving humans. It is therefore crucial to find and fix such errors early on in the development process to ensure a safe and reliable result. Other than that, considering that CPSs are mostly safety critical, it would be beneficial to have a tool, namely UAV-SALIENT helping developers in the early detection of safety-related concerns in user-reported issues as soon as they are reported.

Simulators represent a key asset for CPSs development but their usage and integration could be challenging and make the overall CI/CD process overly slow. The preferred solution in these scenarios is containerization, easy to introduce by setting up families of predefined images specifically tailored for CPS run-time/simulation

needs. Adding new layers on top of available images tailored for CPS development is really simple but, at the same time, can lead to adding unnecessary bloat to them mainly due to developers usually not cleaning the unneeded dependencies resulting in huge packages that are time-consuming to deploy.

The FAT CONTAINER DETECTOR helps determining the extent to which images used for containerizing run-time environments contain unused libraries, hence slowing down the download and usage of images in a CI/CD pipeline. FAT CONTAINER DETECTOR is a standalone tool that can run locally. Based on its output a developer can decide to update the docker image used in the project with the new one produced by the tool. Specifically, given an image, it analyzes its installed libraries and their inter-dependencies, as well as the library usages from source code and build automation scripts and tries to remove dependencies so that the image execution and tests do not fail. As an outcome, it provides developers with the list of libraries that can be removed together with the de-bloated image.

Once having statically identified the presence of performance issues or else symptoms in the code under development that could introduce performance decay, a developer can rely on the tool, i.e., REFACTORING RECOMMENDER able to suggest, for each symptom, a possible refactoring operation. The refactoring tool is able to provide refactoring opportunities for each of the four performance issues statically identified. Specifically, for the *Magical Waiting Number* the goal is to dynamically assign the waiting times according to the hardware component. As regards, instead, the *Hard-coded Fine Tuning* the refactoring solution aims to identify the tuned values for each hardware component before subsequent releases. For the *Fixed Communication Rate*, the refactoring operation ensures that the communication rates between hardware components adapt during the operation of CPS according to the need for communication between components. Finally, for the *Rounding Error* the refactoring assures that the types of the variables do not introduce rounding errors for the values passed to the hardware-related methods.

4.2 Static Analysis

In the following we will provide a brief description of the tools and approaches being developed to help developers in statically identifying (i) issues dealing with the configuration of the CI/CD process, and (ii) performance issues when implementing functionality to be deployed on real hardware devices.

The CI/CD CONFIGURATION ANALYZER extracts facts from the CI/CD configuration files (i.e.,.yaml files) and leverages rules to detect bad choices in the pipeline configuration. Specifically, it is able to detect manual triggering for pipeline stages, build retry, and hiding failures. As regards the former, it is always suggested to avoid including manual tasks, stages, and/or jobs in the CI/CD process, since the main goal of a CI/CD process is to keep the production code in a deployable status at any given time. Missing full automation will make the overall build process non-repeatable, as well as might introduce errors and delay the delivery of code changes to the customers. As regards the detection of the build retry, instead, it would help to reduce the presence of flaky behavior during the execution of the CI/CD process. When configuring and using a CI/CD pipeline for CPS development, it is important to avoid the non-determinism, since it may hinder the development experience, slow down progress, and hide real bugs. Some pipelines address this issue by rerunning a job multiple times after failures. However, this might not only hide an underlying problem but also make issues harder to debug when they only occur sometimes. Finally, when configuring a CI/CD pipeline, developers try to include several stages, each one aimed at spotting a specific type of issue/defect affecting the system under development. Obviously, to identify the defects as soon as they are introduced in the production code, it is important that each stage should be able to fail the build. If this is not the case, developers can miss or ignore the underlying issue, leading to the decay of the CI/CD process in the long term. CI/CD frameworks allow the definition of build stages and/or jobs which outcome do not affect the overall build outcome, this case is identified by the CI/CD CONFIGURATION ANALYZER and notified to developers. To summarize, each time a developer modifies the CI/CD configuration file, the execution of "the linter" is enabled within the build process and it will provide feedback to developers reporting:

- a list containing the name of the jobs allowed to fail, if any;

- a list containing the name of the stages with at least one job that must be triggered manually, if any;
- a list containing the name of the job(s) for which the retry parameter has a positive value (> 0) together with the value aimed at specifying what is the number of attempts scheduled for the job, if any.

Integrating the CI/CD CONFIGURATION ANALYZER within the CI/CD pipeline, it is possible to increase developers' awareness about possible bad practices negatively impacting the overall CI/CD process in place, such as the presence of maintainability issues in the long term, as well as performance issues increasing the time interval between the change and the feedback for developers.

Functional correctness is not the only criteria to measure software correctness. In other words, developers should go one step further beyond functional bug fixes, by enhancing the code towards achieving performance improvements. When coming to CPS development, the performance issues can be further exacerbated due to the complexity of the code being developed, as well as with the need to have simulators for those complex environments to be developed in-house. Once having identified what are the performance antipatterns affecting CPS developed code, it is important to provide an automated framework that identifies those performance issues as soon as they are introduced within the production code. Four are the CPS-related performance antipatterns being statically detected (i.e., PERFORMANCE ISSUES DETECTOR) by means of a set of predefined rules being checked across the code under development. Specifically:

- *Magical Waiting Number* indicates the lack of a suitable waiting time after sending a request to a hardware component. CPSs are heterogeneous systems in which hardware and software components interact. When a component sends a request to a hardware device, they need to wait for the response. The amount of time required by the hardware to respond can vary depending on many factors. This issue might show up when a developer either mistakenly does not consider the potential delays from the hardware side and does not add any waiting time when sending a request to hardware, or sets a fixed global value for the time it expects the hardware devices response.
- *Hard-coded Fine Tuning* occurs when a component-specific value, which is required for physical actions, e.g., drones landing or moving joints of robots, is manually tweaked for CPS performance enhancement. Tuning values with this method can be time-consuming, as well as introduce several performance issues for the upcoming releases. Developers should set these values with logical reasoning or according to an extensive tuning process.
- *Fixed Communication Rate* indicates the need to have minimum latency in operation when multiple hardware and/or software components communicate. Very often the communication rates between CPSs components are assigned statically meaning that the communication rate between two different components is always the same during the operation. This fixed communication rate may lead to performance issues as the data transferred between the components can vary according to the system's status. On the one hand, an extremely high communication rate drives the CPS to higher energy consumption, which negatively impacts the up-time of devices with limited energy resources. On the other hand, setting a low communication rate leads to latency in the communication between components, and thereby efficiency and performance issues.
- *Rounding Errors*. CPSs require numerical values computed dynamically according to inputs from the users and/or the environment to perform some physical operations. These computed numerical values should have the highest precision to assure the accuracy, reliability, safety, and efficiency of the system. One of the common mathematical computation errors that can negatively impact the precision of the computed numbers is rounding error. This type of error refers to scenarios in which one of the numbers is altered to a type with fewer decimals. Rounding Error can lead to both a performance antipattern or a symptom of a functional bug depending on the effect of the computation error.

4.3 Functional testing

In the CPSs context, functional testing mainly relates to testing functional safety, which is the capability of a safety-critical system to correctly respond to inputs and behave in a safely manner. Based on the afore-

mentioned statement, CPS functional testing mainly rely on domain-specific simulators that integrate all the features emulating the environment in which the software has to be deployed later on in production. Simulation is thus a necessity when testing CPS software but the type of simulators being used may vary from company to company. In this scenario, organizations need automated software testing solutions that can be used across CPS software that largely differ in architecture and adopted programming constructs. In other words, there is a need to properly integrate in the DevOps pipeline techniques aimed at (i) automatically derive the inputs to cost-effectively exercise the software under test, and (ii) automatically determine the test outcome (i.e., pass or fail) based on high-level specifications provided by the customers or domain experts. To cope with a wide range of design choices and programming languages, it is possible to rely on a black-box automated testing solution, i.e., TEA (TEST EXECUTION AUTOMATON) that, since CPS software often needs to react to external agents (animated and inanimate ones), relies on reinforcement learning, i.e., learning how to interact with an environment towards a goal (e.g., make the software fail). It is important to note that, TEA is not coupled with any specific simulator while simply requiring that the simulator can be accessible and handled through an external interface. Going on how the tool actually works, once having defined inputs and outputs of interest for the requirement under test, TEA is able to learn how to exercise actions for identifying failures, i.e., it automatically generates episodes to test. Unfortunately, TEA is not able to run without human intervention, meaning that it requires as input the specification of the testing problem (i.e., the requirement under test) by the domain expert for determining the configuration for the reinforcement learning algorithm. The latter is needed only when developers need to test a software component from scratch while they can rely on the already trained model for future software updates. During the training and testing stages, TEA collects run-time data that might be useful to debug the software component under test; more precisely, it collects the initial state, the termination condition, the action that was taken at every step, the time at which that action was taken, the following state, and the reward. The collected data can be further manually inspected for identifying individual failures, i.e., read the log files produced during testing and attempt to recreate an episode by performing the same actions at the same time, but TEA also provides aggregate analysis since, within CPSs, failures are often inevitable given sufficient adverse conditions. The three metrics being provided are:

- the proportion of episodes ending with a failure;
- the average number of steps taken before a failure is observed in failing episodes, reflecting the likelihood for a failure to be observed in reality (i.e., if a few steps are sufficient then it is very likely to be observed, if a long sequence of steps is needed then it is unlikely);
- a cumulative measure of domain-specific adverse conditions.

When talking about CPS development, extensive HiL testing is required before its release. However, HiL testing is often time-consuming, expensive and dangerous. before testing it directly on the hardware components testing is applied relying on simulated environments. Of course the trustworthiness of the results are highly dependent on how WELL the simulators emulate the complex environments. To also help developers with HiL testing, MORLOT (MANY-OBJECTIVE REINFORCEMENT LEARNING FOR ONLINE TESTING) extends the functionality provided by TEA by combining two distinct approaches: (i) reinforcement learning to continuously interact with the environment with the aim of violating requirements and (2) many-objective optimization to solve multiple objectives (i.e., violating requirements) simultaneously, i.e., to efficiently satisfy as many independent requirements as possible within a limited testing budget. Moreover, it is able to work with both modes of testing (software- and hardware-in-the-loop). During its execution, MORLOT collects run-time data of every test case finding a violation and returns all the failing test cases that can be used later on by developers for regression testing. In order to be able to help for HiL testing, MORLOT re-uses the information gained during testing relying on simulators. Going deeper on how and why MORLOT takes advantages of the outcome of the test executions in a simulated environment, it is important to state that HiL testing may often highlight the limitations of the system that were only partially visible within simulated environments, such as the presence of noise and communication delays that affect the system's performance. However, features that have shown to be robust in simulated environments are unlikely to fail only because of noise in HiL; instead, features that have shown to be less robust in simulated environments may actually lead to failures in the presence of noise in the real environments.

The challenges that developers can try to overcome by integrating MORLOT in their CI/CD pipeline for CPS development are mainly two. On the one hand, when dealing with CPSs there are often many safety and functional requirements, possibly independent of each other, that must be considered together. Though one could simply repeat an existing test approach for individual requirements, it is inefficient due to its inability to dynamically distribute the test budget, mainly the time needed to complete the testing phase, over many requirements according to the feasibility of requirements violations. On the other hand, varying dynamic environmental elements, such as weather conditions, during test case execution is key to detect the possibility of requirement violations in real-world usage. Not accounting for such dynamic environments could significantly limit test effectiveness by limiting the scope of the test scenarios being considered.

A different challenge experienced by developers when writing functional test cases for CPSs deals with the appropriate derivation of the test inputs that are usually data coming from real sensors. An inappropriate choice might reduce the effectiveness of the testing phase both in a simulated environment and with HiL. To help developers properly derive the test inputs better representing what usually could occur in a real environment, within COSMOS it has been developed TEASER (SIMULATION BASED CAN BUS TESTING), a tool for simulation-based CAN bus testing that translates data of a Self Driving Car (SCD) obtained from a simulation environment, for deriving test input data for testing the communication through a CAN bus. In doing this, TEASER relies on "emulated" sensor data obtained from multiple different simulation environments assuming that this variability would produce more realistic CAN signals for testing, improving the effectiveness of the test suite.

When developing CPSs, there is a need to cope with the presence of a high number of external factors that in a real environment might reduce the effectiveness of simulation-based testing. To cope with this challenge and to be able to define a set of challenging test inputs highly reflecting what could happen in real scenarios, SURREALIST could be integrated within the CI/CD pipeline. Specifically, SURREALIST is a tool able to replicate field tests in simulation for unmanned aerial vehicles (UAVs) and automatically generate "challenging" simulated test cases in their neighborhood. Specifically, it is a search based approach that analyzes logs of real UAV flights and automatically generates simulation-based tests in the neighborhood improving the realism and representativeness of the simulation-based tests. In doing this, it could be also possible to partially mitigate all the challenges coming from the usage of simulated environments for testing purposes such as the impossibility to deal with real-time properties or else the interaction with the environment.

4.4 Regression testing

One of the challenges experienced by practitioners when using a CI/CD pipeline for CPS development deals with the need to provide quick feedback to software developers in order to be able to identify bugs as soon as they are introduced in the production code. Of course, as soon as the software under test increases their size and their functionality it would become infeasible to run the whole set of test cases for regression with a very limited time budget. To have a CI/CD process that is able to spot bugs quickly it is needed to integrate within the CI/CD pipeline tools for prioritizing and/or selecting the test cases in the regression test suite. As regards the latter, SDC-SCISSOR is a framework that identifies Self-Driving Car (SDC) tests that are unlikely to detect faults in the SDC software under test, thus enabling testers to skip their execution and drastically increase the cost-effectiveness of simulation-based testing of SDCs software. SDC-SCISSOR supports two main usage scenarios: Benchmarking and Prediction. In the Benchmarking scenario, developers leverage SDC-SCISSOR to determine the best ML model(s) to classify SDC simulation-based tests as safe or unsafe. In the Prediction scenario, instead, developers use those model(s) to classify and select newly generated test cases. Other than reducing the overall execution time, SDC-SCISSOR could be used for automatic test cases' generation helping developers with the manually derived set of test cases, as well as increasing the overall level of realism. The tool will provide a report to developers properly highlighting the set of test cases classified as "safe", i.e., the ones that could be manually excluded from the test suite to take control over the overall build execution time.

A different technique that could be used to improve the fault-detection rate of existing test suites deal with test case prioritization, meaning that it will assign high priority to test cases having the highest probability to spot bugs by also accounting for the type of change being submitted by developers. In this scenario, it is possible to integrate, within the CI/CD pipeline, SDC-PRIORITIZE, i.e., a black box test case prioritization helping into achieve cost-effective regression testing in virtual environments. Specifically, SDC-PRIORITIZE ranks test cases so that failing ones are executed first. The tool does not require previous execution traces while defining static features that can characterize safety-critical scenarios in virtual tests.

CPSs exhibit a level of non-determinism in their behavior, meaning that running the exact same inputs under the exact same conditions may result in different outputs and behaviors. Overall,non-determinism in the system has serious ramifications for testing including flaky tests where a single execution of the test inputs is not sufficient to mark the test as passing or failing. Flaky and non-deterministic tests may lead to a false conclusion that a problematic software change does not have a problem or that a good change has problems. Previous investigations have identified that root causes behind flaky behaviour for CPSs are slightly different from the ones usually responsible of the non-deterministic behaviour within traditional software, and developers struggle in properly identifying, as well as removing them from the code. Furthermore, root causes might also vary based on the application domain of the CPS. Within the automotive domain it is possible to rely on SDC FLAKY TEST ANALYZER that, if properly integrated within the CI/CD pipeline can provide feedback on the presence of test cases being non-deterministic and trying to spot out the likely root causes of the non-determinism applied to self-driving car. As regards, instead, the development of unmanned aerial vehicles (UAVs), it is possible to integrate UAV FLAKY TESTS ANALYZER. Specifically, UAVs can behave non-deterministically both in simulation and real environments due to several reasons including but not limited to (i) noise in the sensor outputs, (ii) message delays in the communications between different components, (iii) battery malfunctioning, (iv) changes in weather conditions, and also (v) the presence of randomness involved in the algorithms controlling them. UAV FLAKY TESTS ANALYZER starts by generating “challenging test cases for the UAV” and analyzes the behavior of the drone, executing the exact same test case multiple times.

4.5 Security assessment

Software security testing can be performed by ensuring that documented software misuses are infeasible, and by automatically identifying new software misuses that break the security properties of the software, i.e., security vulnerability testing. Three different challenges can affect an appropriate security assessment for the software under test in a CPS context. First of all, it is challenging to ensure that documented software misuses are infeasible due to the need for ensuring that the system allows only valid interactions (i.e., implements its security requirements) and is free from previously detected, or foreseen, vulnerabilities. A different challenge comes with the need for automated solutions to identify undocumented misuses that break the security properties of the software under test, i.e., the discovery of unknown vulnerabilities through the identification of specific inputs for which the software does not adhere to its security requirements. Last but not least, the third challenge is about the efficient and automated selection of test inputs and oracles. Since security vulnerabilities often manifest with a specific set of inputs, it might be necessary to test the system with a large set of inputs, which might be exacerbated in the case of CPS due to the diverse nature of their inputs. Moreover, in the absence of strategies to automatically verify the output of the system (e.g., by automatically determining the expected output values), exhaustive testing remains infeasible.

MST (METAMORPHIC SECURITY TESTING) is able to automatically identify that documented and unknown, i.e., new software misuses, software misuses are infeasible. In order to do so, the tool requires a set of manually written specifications, i.e., metamorphic relations (MRs), capturing the relations between the output for a valid input and the output for a follow-up input. The follow-up inputs are derived by altering the valid inputs as an attacker would do. Furthermore, the tool is able to automatically generate test cases based on documented software misuses (misuse case specifications in natural language). Specifically, MST performs security testing of the software under test (SUT) by automatically generating inputs and by verifying outputs at the system

level reporting the list of vulnerabilities detected. Going deeper on the tool output, to facilitate adoption it is extremely important to provide information that facilitates the understanding of MR failures. For each failure, MST would be able to provide a report detailing (i) the list of valid inputs used by the MR with the whole set of their actions; (ii) the list of follow-up inputs used by the MR together with their actions; and (iii) the list of actions within valid inputs and follow-up inputs whose outputs had been processed by the MR highlighting the actions leading to an unexpected result.

By integrating MST into a CI/CD process for CPS development it is possible to deal with the oracle problem, i.e., situations where it is extremely difficult or impractical to determine the correct output for a given test input, as well as with the automated identification of test inputs.

Detecting vulnerabilities in a CPS is valuable, but being able to determine potential causes for these vulnerabilities is even more helpful for developers to secure their code. SASVD (STATIC ANALYSIS SECURITY VULNERABILITY DETECTION) is a source code static analysis tool aimed at detecting CPS-vulnerabilities, reporting the likely causes of the specification violation, as well as the vulnerability location in the source code. Specifically, SASVD is based on a specification language by means of which it would be possible to define static aspects of the source code that can be used to express a large variety of security properties. Based on the aforementioned specifications, it determines chains of elements, by analyzing traces obtained during multiple executions of the source code, leading to a specification violation, as well as provides metrics quantifying uncertainties related to the security report so developers can be aware of how much they can rely on the performed analysis. One aspect to take into consideration when providing developers with a set of suspicious code elements needing further investigation is that they usually tend to give up after investigating a given number of spectra. The latter points out how important would be to rely on an appropriate suspiciousness ranking metric aimed at reporting only relevant code elements. For this reason, SASVD integrates multivariate suspiciousness metrics to take into account the effects of combining spectra, as well as general suspiciousness metrics able to capture various statistical dependencies. Another aspect to consider is that developers do not follow the list of suspicious elements sequentially while starting from highly ranked elements. Hence, it is of paramount important to point developers to good starting points. To help with this, SASVD focuses on spectra which are root causes.

By relying onto previous execution traces, when integrating SASVD into a CI/CD process it would be possible to help developers with the low capability of handling real-time properties, specifically in those contexts where HiL testing is highly expensive and only done on the last mile. Furthermore, it helps addressing the challenges dealing with the need to manually specifies test cases for security assessment.

4.6 Continuous Monitoring and DevOps

Run-time Verification is the process of deciding whether an execution, represented as a trace, of some computational system holds some properties captured by a specification, written in a specification language. Once a specification has been written, by relying on specific procedure, it must be decided whether a given trace satisfies that specification. This process can either be performed while the system is still running (online monitoring), or after the execution (trace checking). On the one hand, online monitoring can be useful when one would like to be aware of unexpected system behaviour as soon as possible. On the other hand, trace checking can be useful when one does not need to know about unexpected behaviour soon after it takes place. Trace checking is more straightforward than online monitoring since the relevant information can be found immediately. PECL is a tool helping developers with both online monitoring and trace checking since it determines whether an execution of a given CPS satisfies some specifications provided in a predetermined specification language. The approach can be seen as a dynamically identification of security vulnerability violations by analyzing the execution traces. The main challenge addressable by integrating this monitoring and checking tool within a CI/CD pipeline for CPS development is the one dealing with the high complexity in specifying the oracle for what concerns security requirements that are mainly written by spotting behaviour that one does not expect to occur within the system under development.

In the following we will provide a brief description of the tools and approaches being developed to help developers in properly setting and customizing the CI/CD pipeline for CPS development.

In the context of CPSs, the usage of CI/CD is highly important mainly due to the complex environments, the intensive usage of simulators, and the diverse targeted hardware devices. The aforementioned properties introduce difficulties in automating builds in such complex environments, as well as challenge the deployment of the software onto hardware devices and/or simulators. To help developers overcoming some of the challenges dealing with setting up and maintaining a CI/CD pipeline, within COSMOS have been developed a tool, namely BUILD LOG ANALYZER that analyzes the CI/CD at run-time by mining its logs to detect CI/CD misuses or decay. Specifically, it verifies (i) the extent to which the release branch is broken, (ii) whether jobs are skipped because of their failures, (iii) whether the build time exhibits an uptrend, or the build exhibits a duration significantly longer than previous builds, (iv) improper cache handling, (v) inconsistent build behavior over multiple environments, and (vi) suboptimal build ordering due to frequently-failing stages occurring at a later stage of a build. The BUILD LOG ANALYZER rely on both static (CI/CD configuration file) and dynamic (i.e., history of the build logs in a specified time interval) information. Going deeper on the issues the tool is able to detect:

- *A stable release branch is missing.* Lacking proper management of features development branches might hinder the overall CI/CD pipeline efficiency and effectiveness. In other words, the presence of a broken release branch that is not fixed timely might prevent the effectiveness of the overall CI/CD process, i.e., lack of a releasable version of the system under development.
- *Pipeline steps/stages are skipped arbitrarily.* It is possible for developers to skip some "problematic" pipeline steps/stages, hiding the actual behavior of the CI/CD process and, more importantly, making untrustworthy the released version. The latter might be due to the need of having a green build status. The aforementioned behavior goes against the CI/CD principles since developers will only delay the discovery of potential problems in the developed code.
- *Build time too long.* A slow build produces waiting times for developers and adds overhead to the CI process. For CPSs, the problem can be further exacerbated upon deploying and executing software on simulators or HiL.
- *Inappropriate cache handling.* Caching strategies are very often used to store the content that does not change among subsequent builds and reuse them to speed up the overall CI/CD process. It becomes useful warning developers in the presence of jobs having a "long" duration for which probably the enabling of the caching feature could produce a saving of the overall CI/CD execution time.
- *Different jobs consistently give a different outcome.* For CPS development it is important to rely on jobs with the goal of having the same (or slightly different) build process running on multiple different environments (e.g., different simulators and/or also different versions for the hardware devices). In this scenario, it is desirable to discriminate whether or not the root cause behind the failure is related to the actual code change pushed to the repository, or else there is a specific "problematic" environment consistently marking the overall build outcome as failed. In this way, it would be possible to prevent developers from completely losing faith in the output provided by the CI/CD process, as well as to check the quality of their proposed changes to identify possible issues and/or drawbacks as soon as they are introduced into the system.
- *Build stages are not properly ordered.* It is important to properly define the appropriate order of execution of the different stages to avoid the situation where a stage responsible for the build failures is always executed towards the end of the CI/CD process, leading to wasting resources and time needed to provide feedback.

Each time a new build log has been provided by the CI/CD framework, the BUILD LOG ANALYZER will provide detailed feedback to developers reporting: (i) the maximum number of consecutive failures occurring on the release branch together with the percentage of failures experienced on it; (ii) the number of tests being arbitrarily skipped (if any), and the name of the jobs that are hidden from the configuration file (if any); (iii) whether the build execution time is in line with what is considered acceptable by the organization, or else it is

increasing; (iv) the name of the jobs (if any) whose execution time is higher than the median execution time in the same CI/CD pipeline for which the cache feature is not enabled; (v) the list of jobs (if any) which outcome is consistently failing with respect to the other jobs included in the CI/CD pipeline; and (vi) the name of the stages (if any) that are consistently responsible for the failure that are executed at the end of the CI/CD process.

CPS development requires the use of either simulators or HiL during various stages of the development activity, and in particular upon testing the systems. One of the challenges experienced when setting a CI/CD pipeline is to cost-effectively use the limited number of simulators and hardware devices. It is important to optimize the resources' usage by properly allocating tasks/jobs across simulators and/or HiL within the CI/CD pipeline. To help with the overall optimization of the resources being available without hindering the trustworthiness of both developers and customers about the software being released, we can rely on two different tools, i.e., JOB PRIORITIZER and JOB SMART ALLOCATOR. The former can be used to predict the schedule of each job included in the CI/CD process, while the latter aims at optimizing the allocation of simulators and hardware devices taking into account dependencies between different tasks/jobs.

As regards the JOB PRIORITIZER, given an incoming build, it may or may not be necessary to execute it immediately, i.e., if a build helps discovering faults and its execution would be fast, then it may be worthwhile to execute it, otherwise, the build jobs could be rescheduled in a periodic build. Specifically, the tool will estimate the job execution time and based on it will determine the appropriate schedule to be assigned to each job included in the CI/CD process. So the outcome is a modified version of the CI/CD configuration file that can be approved or not by a developer before enacting the overall CI/CD process.

As regards the JOB SMART ALLOCATOR, if multiple resources (e.g., HiL or simulators) are available, a monolithic build could be decomposed into multiple, parallel ones. For instance, if a build contains several tests, the tests could spread over the available machines (HiL/simulators). Using the tool it would be possible to minimize the total build time while maximizing the testing effectiveness by optimizing the usage of simulators and hardware devices. In this way it is possible to obtain a uniform allocation of jobs and if, simulators and/or hardware devices are rented from third-party, minimize the overall cost. Given a set of resources being available, the tool determines whether some jobs can be split and parallelized, as well as the allocation of build jobs onto different simulators and/or hardware devices.

Setting up and maintaining a CI/CD pipeline requires knowledge and skills often orthogonal to those entailed in other software-related tasks. The intrinsic CI/CD requirements and underlying technology of a given project may make the configuration of the CI/CD process extremely hard, e.g., a system needs to be deployed and tested on different operating systems or even embedded devices. Specifically, the structure and code of a CI/CD pipeline may be less regular and repetitive than traditional software system since it mixes up very specific scripting elements (e.g., related to configuring a server) with some more recurring and regular reusable elements up to natural language elements. Furthermore, a CI/CD pipeline contains several context-specific elements, e.g., paths of installation directories, or URLs of resources to download. To help overcoming the above challenges, developers might rely on a tool for automated completion of CI/CD workflows, i.e., WORKFLOW COMPLETION leveraging the Text-to-Text Transfer Transformer (T5) model. The proposed tool can recommend work-flow completions in different modes that mimic how a developer may implement the workflow, i.e., (i) suggesting the next statement, or (ii) helping to complete a job with implementation elements once the developer has defined, in plain English, what the job should do (e.g., Yarn install).

5 Conclusions

This deliverable presents a methodology for instantiating, configuring and maintaining CI/CD pipelines for CPS development. Starting by summarizing the main challenges dealing with both the overall CI/CD process in place and the concrete CI/CD pipeline in use within the organization, we will provide a set of guidelines to apply CI/CD for CPS development. Specifically, the deliverable provides a tentative CI/CD pipeline that can be used to address some of the challenges experienced by developers by properly identifying how and

where it is possible to integrate the tools and/or approaches being developed within COSMOS (see Figure 2 and Table 7). Specifically, the methodology being presented places COSMOS approaches and tools around a CI/CD pipeline, i.e., leverages COSMOS techniques and tools to create a CI/CD pipeline for CPS. In doing this, it maps the challenges coming from deliverable D3.2 [9] to the stages mainly included in a CI/CD pipeline for CPS and to COSMOS approaches and tools, where available.

References

[1] MISRA:2012 Guidelines for the use of the C language in critical systems. https://www.academia.edu/40301277/MISRA_C_2_012_Guidelines_for_the_use_of_the_C_language_in_critical_systems.pdf. Accessed: 2021-08-30.

[2] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for cyber physical system. In *Testing Software and Systems*, pages 194–207, Cham, 2015. Springer Intern. Publishing.

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

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

[5] Ron Bell. Introduction to iec 61508. In *ACM International Conference Proceeding Series*, volume 162, pages 3–12. Citeseer, 2006.

[6] Frederick P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

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

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

[9] COSMOS Project. Catolog of good and bad practices when applying ci/cd in cpsos development., 2021.

[10] Amit Deshpande and Dirk Riehle. Continuous integration in open source software development. In *IFIP International Conference on Open Source Systems*, pages 273–280. Springer, 2008.

[11] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. CARLA: an open urban driving simulator. In *1st Annual Conference on Robot Learning, CoRL 2017*, volume 78 of *Proceedings of Machine Learning Research*, pages 1–16. PMLR, 2017.

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

[13] Paul M. Duvall. Continuous integration. patterns and antipatterns. *DZone refcard #84*, 2010.

[14] Paul M. Duvall. Continuous delivery: Patterns and antipatterns in the software life cycle. *DZone refcard #145*, 2011.

[15] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A Ernst, and Margaret-Anne Storey. Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering*, 2021.

[16] Erik Flores-García, Goo-Young Kim, Jinho Yang, Magnus Wiktorsson, and Sang Do Noh. 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*, volume 592 of *IFIP Advances in Information and Communication Technology*, pages 238–244. Springer, 2020.

[17] Martin Fowler and Matthew Foemmel. Continuous integration. https://web.archive.org/web/20200522100521/https://martinfowler.com/articles/originalContinuousIntegration.html. Accessed: 2021-11-21.

[18] Heinz Gall. Functional safety iec 61508 / iec 61511 the impact to certification and the user. In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 1027–1031, 2008.

[19] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci. *IEEE Trans. Software Eng.*, 46(1):33–50, 2018.

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

[21] Philipp Helle, Wladimir Schamai, and Carsten Strobel. Testing of autonomous systems - challenges and current state-of-the-art. *INCOSE International Symposium*, pages 571–584, 2016.

[22] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 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*, 2017.

[23] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 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)*, 2016.

[24] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, 2011.

[25] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. Quality metrics and oracles for autonomous vehicles testing. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 194–204. IEEE, 2021.

[26] Yujuan Jiang and Bram Adams. Co-evolution of infrastructure and source code: An empirical study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 45–55. IEEE Press, 2015.

[27] Suzette Johnson, Harry Koehneman, Diane LaFortune, Dean Leffingwell, Stephen Magill, Steve Mayner, Avigail Ofer, Robert Stroud, Anders Wallgren, and Robin Yeman. *Industrial DevOps - Applying DevOps and Continuous Delivery to Significant Cyber-Physical Systems*. National Academies Press, 2017.

[28] Nidhi Kalra and Susan Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 12 2016.

[29] Eero Laukkanen, Maria Paasivaara, and Teemu Arvonen. Stakeholder perceptions of the adoption of continuous integration–a case study. In *Agile Conference (AGILE)*, pages 11–20. IEEE, 2015.

[30] Ivano Malavolta, Grace Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. 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, page 31–40, New York, NY, USA, 2020. ACM.

[31] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. Continuous integration applied to software-intensive embedded systems–problems and experiences. In *International Conference on Product-Focused Software Process Improvement*, pages 448–457. Springer, 2016.

[32] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 141–150. IEEE.

[33] Ade Miller. A hundred days of continuous integration. In *Agile, 2008. AGILE'08. Conference*, pages 289–293. IEEE.

[34] Vuong Nguyen, Stefan Huber, and Alessio Gambi. Salvo: Automated generation of diversified tests for self-driving cars from existing maps. In *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 128–135. IEEE, 2021.

[35] Helena Holmstrom Olsson, Hiva Alahyari, and Jan Bosch. 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, pages 392–399, Washington, DC, USA, 2012. IEEE Computer Society.

[36] Heejong Park, Arvind Easwaran, and Sidharta Andalam. Challenges in digital twin development for cyber-physical production systems. In *Cyber Physical Systems. Model-Based Design*, pages 28–48, Cham, 2019. Springer International Publishing.

[37] Akond Rahman and Laurie A. Williams. Characterizing defective configuration scripts used for continuous deployment. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 34–45. IEEE Computer Society, 2018.

[38] Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. Synthesizing continuous deployment practices used in software development. In *Agile Conference (AGILE), 2015*, pages 1–10. IEEE.

[39] Hendrik Roehm, Jens Oehlerking, Matthias Woehrle, and Matthias Althoff. Model conformance for cyber-physical systems: A survey. *ACM Transactions on Cyber-Physical Systems*, 3(3):1–26, 2019.

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

[41] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 189–200.

[42] Hesham Shokry and Mike Hinchey. Model-based verification of embedded software. 2009.

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

[44] Sirasak Tepjit, Imre Horváth, and Zoltán Rusák. 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):527–541, 04 2019.

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

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

[47] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of ci build failures: An open source and a financial organization perspective. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 183–193.

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

[49] Fiorella Zampetti, Damian Tamburri, Sebastiano Panichella, Annibale Panichella, Gerardo Canfora, and Massimiliano Di Penta. Continuous integration and delivery practices for cyber-physical systems: An interview-based study. *ACM Trans. Softw. Eng. Methodol.*, 32(3), apr 2023.

[50] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One size does not fit all: An empirical study of containerized continuous deployment workflows. 2018.