Microservices for Continuous Deployment, Monitoring and Validation in Cyber-Physical Systems: an Industrial Case Study for Elevators Systems

Aitor Gartziandia*, Jon Ayerdi†, Aitor Arrieta†, Shaukat Ali‡, Tao Yue‡, Aitor Agirre*,
Goiuria Sagardui† and Maite Arratibel §
Ikerlan*, Mondragon University †, Simula Research Laboratory ‡, Orona§
*{agarciandia, aagirre}@ikerlan.es, †{jayerdi,aarrieta,gsagardui}@mondragon.edu,
‡{shaukat, tao}@simula.no, §marratibel@orona-group.com

***Abstract*—Cyber-Physical Systems (CPSs) are systems that integrate digital cyber computations with physical processes. The software embedded in CPSs has a long life-cycle, requiring constant evolution to support new requirements, bug fixes, and deal with hardware obsolescence. To date, the development of software for CPSs is fragmented, which makes it extremely expensive. This could be substantially enhanced by tightly connecting the development and operation phases, as is done in other software engineering domains (e.g., web engineering through DevOps). Nevertheless, there are still complex issues that make it difficult to use DevOps techniques in the CPS domain, such as those related to hardware-software co-design. To pave the way towards DevOps in the CPS domain, in this paper we instantiate part of the reference architecture presented in the H2020 Adeptness project, which is based on microservices that allow for the continuous deployment, monitoring and validation of CPSs. To this end, we elaborate a systematic methodology that considers as input both domain expertise and a previously defined taxonomy for DevOps in the CPS domain. We obtain a generic microservice template that can be used in any kind of CPS. In addition, we instantiate this architecture in the context of an industrial case study from the elevation domain.**

***Index Terms*—Microservices, DevOps, Cyber-Physical Systems**

I. Introduction

Cyber-Physical Systems integrate digital cyber computations with physical processes [14]. These systems are inherently complex, and their lifecycle can last up to 30 years in sectors such as railway or elevation [6]. In these systems, an increasing trend is to implement most of the functionalities through software. During the life-cycle of these systems, the software continuously evolves due to hardware obsolescence, requirement changes, vulnerabilities, bug corrections, etc. Consequently, this evolution requires reliable and automatic engineering methods for developing and operating CPSs.

With existing engineering practices for CPS, releasing and deploying new software versions is a time-consuming and error-prone activity. This is mainly due to the impossibility of thoroughly testing the software in a real environment. Furthermore, the deployment process itself is complex, as it is highly important to ensure that the CPS will be in a safe state when the software is updated. Besides, these systems often operate in dynamic and uncertain environment, what makes appropriate self-healing and recovery mechanisms necessary. These problems can be partially solved by implementing design-operation continuum methods for the software development life-cycle, instead of relying on traditional software development methods (e.g., the V model). Nevertheless, to achieve this in the CPS domain, radically new solutions to overcome the limitations of today's CPS development processes need to be adopted.

As an alternative, in the context of the Adeptness H2020 project [1] a reference architecture was proposed to enable Design-Operation Continuum activities in CPSs. The contribution of this paper is instantiating this architecture in an industrial case study from the elevation domain. A system of elevators is a complex CPS where all the aforementioned problems frequently arise. By using this architecture, we foresee significant enhancements in the software development, significantly reducing the software development cost while increasing its quality.

The rest of the paper is structured as follows. Section II presents the industrial case study in which we applied the architecture and the problems that they face. We explain the methodology for developing the architecture in Section III. Section IV presents the architecture based on microservice. Section V presents the prototypical implementation and a qualitative evaluation. We position our paper with the state-of-the-art in Section VI. Lastly, we conclude the paper and discuss the future avenues in Section VII.

II. Case Study and Problem Representation

Orona is a company dedicated to the designing, manufacturing, installing, and maintaining elevators, escalators and moving ramps. Elevator installations are complex CPSs that provide service to the passengers, considering the passengers' active passenger calls and the elevators' status. An overview the different elements of the CPS are depicted in Figure 1.

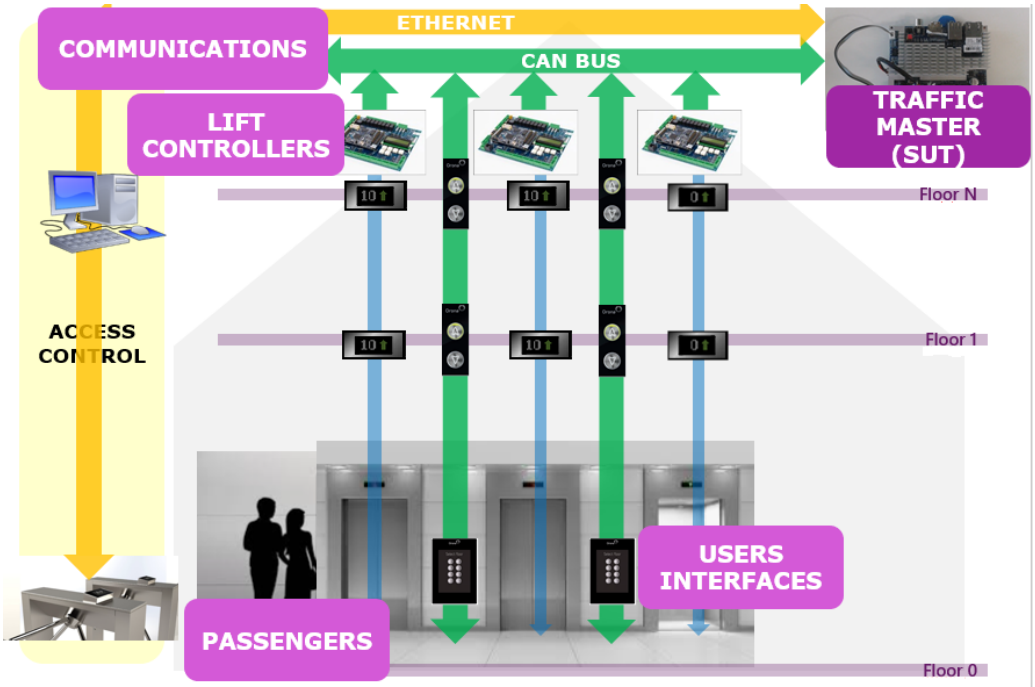

Fig. 1: Overview of the architecture of the Elevators installation

User interfaces allow passengers to introduce calls to the system in different ways. Conventional user interfaces consist solely on Up and Down call buttons, where only the floor on which the passenger is located and the direction of the journey are provided. Inside the elevator, the call panel allows introducing the destination floor of the passenger. On the other hand, destination input devices allow passengers to introduce the destination floor when making a call and then, passengers receive information about which elevator is going to attend them. The elevator call panel inside the elevator is not required in this case. There are also signalling panels indicating the current floor of the elevator and the expected journey. The interaction with the passenger can be extended with access control systems as well.

Each elevator is managed by a controller, which is responsible for the vertical and horizontal (doors control) movements of the elevator. When an elevator receives a landing or a car call (i.e., a call from inside the cabin), the controller decides the order of the stops and the opening and closing of the doors in each floor to attend all the calls by considering different information, including traveling direction, floor position, already assigned calls, etc. This information flows from one device to another through a (vertical) CAN bus.

The traffic master is the component that coordinates the user interfaces with the lift controller. It receives the information from access control devices through Ethernet and checks whether a passenger has the right to issue a particular call. It receives the passenger call from the landing call panels through the (horizontal) CAN bus, and the information of the status, position, etc, from each lift controller. With this information, the traffic master decides which is the best elevator to attend every call, considering different criteria, such as minimising the Average Waiting Time (AWT), the Journey Time (JT) or energy consumption[1]. Finally, the traffic master notifies the lift controller about the assigned call and indicates the assigned lift to the passengers.

This architecture does not provide support for design-operation continuum methods. When the traffic study is performed, there is usually a lack of real and precise data to adequately configure the system. In operation, automatic feedback mechanisms to improve the configuration and detect problems or unknown conditions are lacking. Consequently, when problems or unknown conditions arise in an installation (e.g., degradation in the AWT), the building owner is responsible for communicating the problem to Orona. Validation and deployment of the system are also semi-manual. Regarding validation, information from the operation is not accessible, so it is not possible to reproduce real situations in the laboratory, and the decision of whether a test case has succeeded or not is manual. Regarding deployment, the maintainer is responsible for configuring and updating new versions in the installation, which is also a manual process. In this paper, we present the extension of the architecture to support automatic deployment, continuous monitoring, and validation.

III. Architecture Development Methodology

Before designing the architecture, we defined a methodology that would enable defining the architecture systematically, considering the benefits of microservice architectures over

[1]The AWT is the average time that passengers wait until they enter in the lift. The JT is the average time that passengers wait to reach their destination. Both metrics are used to measure the performance of elevators systems.

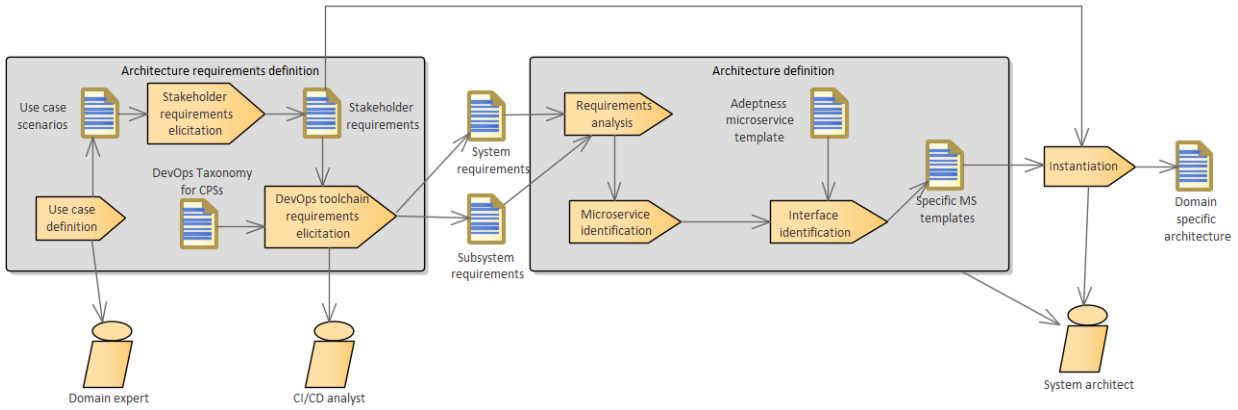

Fig. 2: Methodology for developing the microservices-based architecture and its instantiation to the Elevation domain

monolithic applications [18] and the need for domain awareness when developing an architecture [12]. The architecture development methodology is shown in Figure 2, and it consists of a total of six main steps:

- Use-case definition: First, there was a need to define the use-case scenarios that Orona wanted to handle. To this end, a domain expert defined a set of use-case scenarios, which can be found in [3].
- Stakeholder requirements: With these use-case scenarios, the stakeholder's requirements were elicited. A total of 56 requirements were elicited by an elevation domain expert, which are accessible in [2].
- DevOps toolchain requirements elicitation: By having as input a set of stakeholder requirements and a DevOps taxonomy for CPSs, developed in our previous work [6], two types of requirements were elicited: system requirements, which are those specific requirements from the overall architecture needed to satisfy the stakeholder requirements and subsystem requirements, which are those requirements specific to the subsystems (explained in the following section of the paper) necessary to satisfy system requirements. All these requirements along with the test cases that will be executed to validate them can be found in [2].
- Requirements analysis and microservice identification: With the elicited requirements, a first analysis was performed by a system architect, and a series of microservices were identified.
- Interface identification: For each microservice, the different interfaces were defined and integrated with the Adepteness microservice template, which is available for both C and Python.
- Instantiation: The last step referred to the instantiation and integration of all these templates.

IV. Architecture Based on Microservices

The HORIZON2020 Adeptness project [1] has proposed a microservice based architecture that will allow DevOps practices to be adopted in the context of CPSs. Microservices permit building a flexible architecture where services can be reused in different life-cycle stages and hardware, seamlessly deploying new services to all the installations and scaling the system. Each microservice within the proposed architecture shall be responsible for a specific well-defined function in the life-cycle of a new software release, and shall provide different lightweight communication mechanisms. Each microservice will provide both synchronous (i.e., HTTP) and asynchronous (i.e., MQTT) communication, offering common interfaces for every microservice within the system and custom interfaces for microservices with specific roles.

A. Common interfaces

All microservices within the architecture provide a set of basic asynchronous and synchronous communication endpoints, regardless to the role of the microservice. These endpoints offer basic information about the execution status, health and performance. The synchronous interfaces allow other services to request microservices' health status, while asynchronous interfaces allow microservices to publish relevant data without knowing the receiver of the messages. The following interfaces are provided by the template developed within the HORIZON2020 Adeptness project [1].

1) Synchronous communication:

- /adms/v1/ping [GET]: Ping service to check that the service is alive. Returns an empty 200 response if the microservice is working correctly.
- /adms/v1/info [GET]: Provides basic information about the microservice. It returns a JSON object containing the microservice ID and microservice role within the architecture.
- /adms/v1/performance [GET]: Provides CPU and memory usage metrics. It returns a JSON object containing the free and allocated memory and the CPU usage.
- /adms/v1/status [GET, PUT]: Permits getting or changing the execution status of the microservice. GET calls to this endpoint will return a JSON object containing the status of the microservice. Changes to the microservice status will be performed by sending a JSON object with the

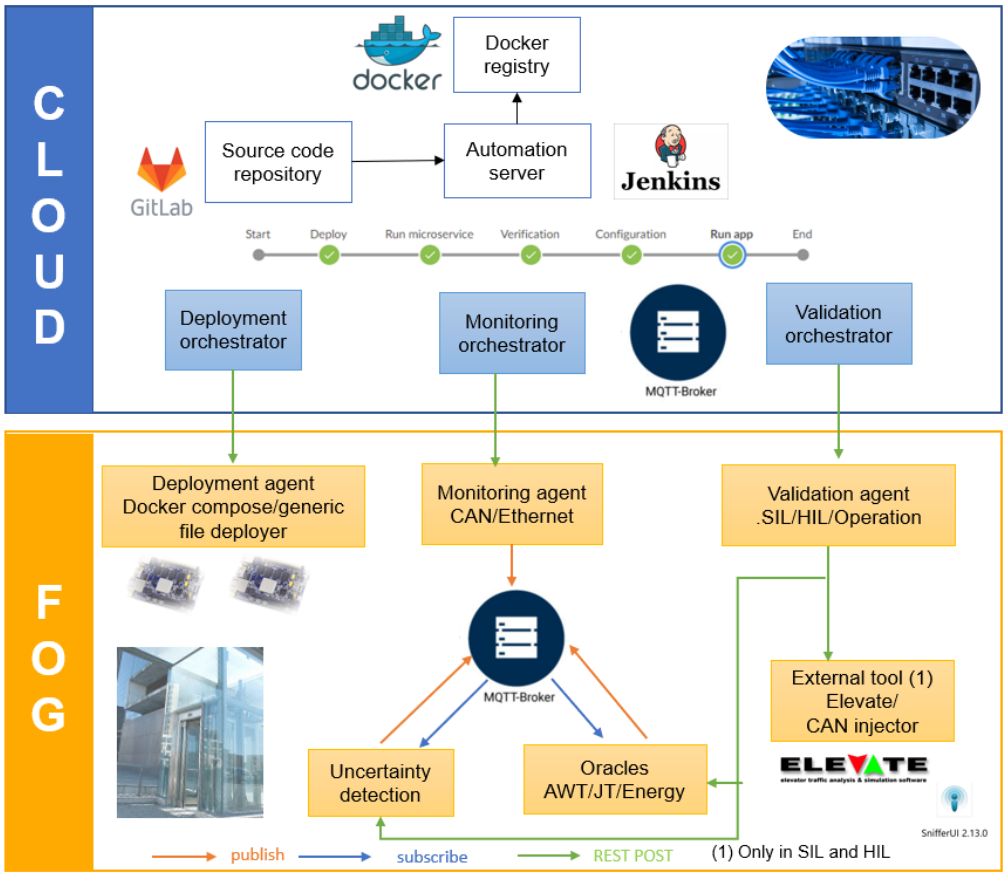

Fig. 3: Overall overview of the microservice-based architecture

desired state. The possible states for the microservice are "Ready" and "Running".

2) Asynchronous communication:

- /adms/v1/discovery [PUB]: On microservice launch, the microservice publishes a hello message in this topic including the identifier, microservice role and its MQTT and REST endpoints, defined as a JSON object.

B. Architecture – Instantiation in the use-case

In Figure 3 we present the instantiation of the Adeptness microservice architecture for ORONA. In particular, we provide microservices for continuous deployment, monitoring and validation, recovery and uncertainty detection.

The main subsystems composing the architecture for ORONA are the following:

1) Automation server: The automation server is in charge of the orchestration of the tasks to be performed by the different subsystems. It interacts with the source code repositories to monitor any changes on the deployment, monitoring or validation plans. When a new plan is updated, the automation server performs the actions to generate the required artifacts, stores the generated Docker images in the Docker registry and pushes the configurations or plans to each subsystem.

2) Deployment subsystem: The deployment subsystem is responsible for downloading, and eventually decompressing and executing, the different microservices and artifacts needed to perform the validation in each of the targets or edge nodes. The deployment subsystem executes a deployment plan and must be aware of the status of the deployment in each node. The plan contains the information regarding the components to be deployed, the repository where they are located in, and the node(s) where they should be deployed. The deployment subsystem is capable of deploying two different types of components, containerized microservices and generic files, and it is composed of two different microservices:

- Deployment orchestrator: The deployment orchestrator receives the deployment plan from the automation server and parses the plan in order to execute it. There is only one instance of this microservice within the architecture and it is usually located in the cloud. The orchestrator sends the deployment instructions to the deployment agents installed in each node by MQTT.
- Deployment agent: The deployment agents must be installed in each edge node to perform the actual deployment of the necessary artifacts. Since there are two different types of components that may be deployed, two types of deployment agents have been designed: a docker-

compose based deployer to deploy docker containers, and a generic deployer to deploy any kind of file, e.g. an executable file, a library, or a zip file. In the latter case, the deployer can perform the actual deployment of the zip file, decompress it, and execute the selected executable file.

3) Monitoring subsystem: The monitoring subsystem supports the configuration of the monitors according to a monitoring plan. This plan specifies the source (physical interfaces, file system, ...) to obtain the data from as well as the value extraction mechanism. This subsystem provides access to telemetry data retrieved from different sources so that other subsystems can subscribe to this data and use it to take decisions. This subsystem consists of two microservices:

- Monitoring orchestrator: This microservice, deployed in the cloud, handles the parsing of the monitoring plan sent from the automation server, and configures all the monitoring agents indicated in the plan accordingly through their HTTP API. The plan specifies the parameters that each monitoring agent needs to specify the actual data source connection parameters (e.g., the CAN baud rate) and the variables to monitor.
- Monitoring agents: The monitoring agents, deployed at the edge nodes, are responsible for reading the operational variables from the different sources and publishing them asynchronously. The monitoring agents can be configured through a common HTTP API, which allows the specification of the variables to be monitored (name and needed parameters to obtain the data) and optionally, the configuration of subscriptions. The concept of subscription is similar to OPC-UA, i.e., a group of variables that are notified asynchronously as events, with the same publishing rate. In this sense, a service (e.g., an oracle) that needs to be notified about the changes of a set of variables can configure a subscription, specifying the publishing rate for those variables.
  There are specific monitoring agents for different data sources. In the case of Orona, two different monitoring agents are used:
  - CAN monitor: The CAN monitor allows the monitoring of the operational variables shared through a CAN field-bus, which may be configured through an HTTP API. For each variable to be monitored, three parameters must be configured: (1) the name of the variable, (2) the identifier of the CAN frame where the variable is published, and (3) the mask to be applied to the frame to actually read the variable. Then, when the monitor starts, it begins to publish the variables asynchronously through MQTT, following the standardized senML[2] payload format.
  - Instrumented code monitor: This is a special monitor type that supports the monitoring of variables that are not exported in any field bus but are needed by the oracles to raise a verdict, for instance, the internal variables of the traffic algorithm that are usually inspected in debugging mode. To do so, a library which publishes code variables through MQTT has been developed. The developer can use it to publish the internal code variables needed by the oracles into the MQTT broker, in the same senML format used by the rest of the monitors.

4) Continuous Validation subsystem: The continuous validation subsystem supports verification and validation activities at Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), Hardware-in-the-Loop (HiL), and Operation. At the MiL test level, the software that controls the physical part of the CPS is a model. At the SiL test leve, this model is replaced by executable software. At the HiL test level, the software is integrated with the real-time infrastructure (e.g., real target processor and operating system) and the physical part emulated within a real-time test bench. For the SiL level, Orona uses Elevate, a domain specific simulator for validation. At the HiL level, a hybrid infrastructure where some components are real and others virtual is used. At this test level, the tests are performed in real-time, using all the real infrastructure, including real communication buses and a real-time operating system. The main microservices used for the continuous validation subsystem for Orona's case are the following:

- Validation orchestrator microservice: Located in the cloud, the validation orchestrator manages the execution of a validation plan by communicating with the validation agents. A validation plan can require validations at different test levels.
- Validation agents for SiL, HiL and Operation: these microservices launch validations at the SiL and HiL test environments, as well as in production installations. For the execution of a validation, child test oracles that provide the verdict are activated. Validation agents in SiL and HiL also manage the tools required for simulating test inputs. When an oracle provides a verdict, it notifies the validation orchestrator microservice.
- Oracle microservice: This microservice encompasses a set of test oracles that validate that the CPS behaves as expected. Many of these test oracles are based on domain-specific Quality-of-Service (QoS) measures that are collected from the monitoring microservices. Among these QoS measures, for the elevation domain, the most important ones are the Average Waiting Time (AWT), the Journey Time (JT) and the energy consumption. Each of these test oracles provide a verdict that indicates to which extent the CPS behaves as expected. Different test oracles have been developed, such as those based on metamorphic relations for the SiL and HiL test levels [7], and some based on machine-learning that predict the maximum AWT and JT a system of elevators should have at each moment.
- Uncertainty detection microservice: This microservice supports the automated detection of unforeseen situations in the different life-cycle stages of CPSoS using data

[2]https://tools.ietf.org/html/rfc8428

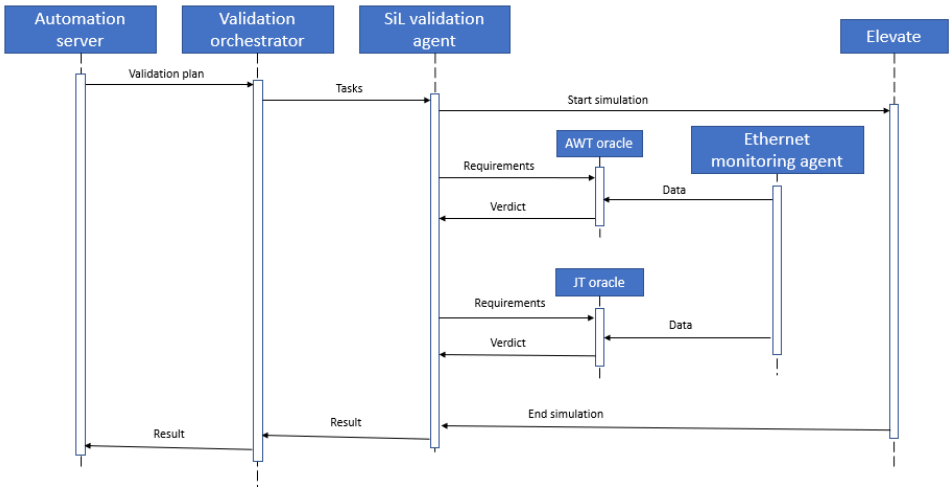

Fig. 4: Overview of the validation process at SiL level of the prototype

from both operation (e.g., live data) and design time (e.g., test logs) with passive and active machine learning techniques. This service supports the validation microservice with uncertainty related test oracles that will be learned from data. Various uncertainties exist in the elevator use case, such as (1) *Passenger data*: Examples include when a passenger arrives at which floor? How much a passenger weighs? (2) *Environment*: Examples include delays in hardware such as motor delay and levelling delay; (3) *Outputs/Quality of Lift Services*: Examples include uncertainties in waiting and transit times.

- External tool microservices for SiL and HiL test levels: This microservice allows launching domain-specific tools required to handle the execution of tests. Two external tool microservices have been instantiated for testing Orona's dispatching algorithm: (1) Elevate (a domain specific simulation tool) that is used for SiL validations and (2) a CAN Bus frames injector for HiL validations.

V. Prototype Implementation and Preliminary Evaluation

To analyse the benefits of the architecture, a preliminary prototype of the architecture for SiL validation in Orona has been developed[3], by setting a pipeline structure in Jenkins. This pipeline starts with the deployment of the components from a deployment plan. If the deployment is successful, the pipeline continues by configuring and starting the monitors, and finally the validation microservices are configured and executed to obtain a verdict. A Docker image for each microservice has been built and pushed to a Docker registry, particularly to the Gitlab container registry.

The deployment pipeline is continuously querying the repositories for changes in the deployment plan, and whenever a change is detected, the deployment pipeline is launched. The deployment pipeline launches a slave agent that logins to the Gitlab container registry and fetches the latest versions of the docker images for the specified components. These images are then started and configured to forward different ports on the slave agent node. The deployment pipeline waits for readiness of every launched microservice by calling to their /adms/v1/ping REST endpoint. The deployment plan consists of a JSON formatted file where each node, identified by its IP address, is assigned one or several components. When the microservices are ready, the monitoring pipeline is called as a build step, or in case of failure, docker containers are stopped and cleaned.

The monitoring pipeline configures the monitoring microservice, by setting the output topic where different values will be published through the HTTP API. When the microservice is ready, the publication of values starts through another call to the HTTP API. If every call succeeds, the validation pipeline is launched as a build step. As in the previous pipeline, on failure, the environment is cleaned and docker containers stopped.

Similar to the monitoring pipeline, the validation pipeline sets up the validation microservice by setting up the input topic where the monitoring microservice is leaving its values, and the conditions that will be evaluated to raise a verdict for the evaluation. After the microservice is configured and started, the validation takes place and the pipeline continuously polls the microservice for a verdict. If the verdict has been marked as passed, the pipeline will succeed, and will fail otherwise. Figure 4 shows an overview of the process.

Table I shows a qualitative overview of the benefits that the proposed architecture can bring during the DevOps activities in Orona. At the deployment level, the automation of the tasks that Jenkins brings reduces effort and saves time for developers by reducing the manual tasks, such as executable generation and deployment for different stages. At the monitoring level, the architecture allows continuously monitoring data from the different sources at all stages, gaining more understanding of the system while reducing the effort. Finally, at the validation

[3]A video of the prototype is available at https://youtu.be/uoq9n9k4kgc

TABLE I: Expected benefits of the architecture in the Deployment, Monitoring and Validation activities for different life-cycle stages in Orona

DEVOPS ACTIVITIES	CURRENT PROCESS	WITH MICROSERVICE ARCHITECTURE	EXPECTED BENEFITS
Deployment			
(SiL) Generate the dll for the domain specific simulator (SiL) Copy files for the simulator (HiL) Compile for the target (HiL) Deploy in the target	Manual compilation & copy	Jenkins pipeline	Not dependence on developer Automatic trigger of the compilation and deploy
(Operation) Deploy in the real installation	Manual deployment by the maintainers	Remotely and automatically deploy a new software release	Effort saving by automatically deploy a new software version. Control over the configuration of the release
Monitoring			
(SiL) Traces of the simulation tools (SiL, HiL, Operation) Traces in the code (HiL, Operation) Monitor the communication buses	Traces are recorded in a txt file Data provided by the simulator in excel and word CAN frames recorded on demand	Data from the code, the simulators and the communication buses will be published by MQTT	Effort saving in analysis of problems in installations Continuous remote monitoring
Validation			
(SiL, HiL) Define test cases: Unitary and QOS (i.e. AWT)	Unitary test cases manually defined QoS test cases from theoretical profiles	Unitary test cases manually defined Automatic profiles from real data of installations.	Test cases defined from real profiles more likely to reproduce real problems
(SiL/HiL/Operation) Execute the validations	Manual configuration of installations Manual trigger in SiL/HiL Manual validation in operation by the maintainer	Set of available configurations for SiL/HiL Automatic Jenkins pipeline for SiL/HiL Continuous validation in operation	Effort saving by automatically validate software release Increase the number of bug detected by continuously validate the software even in operation
(SiL/HiL/Operation) Decide the verdict for the validation	Manual	Reusable oracles	Increase the number of bugs detected Minimise dependency on individuals
(SiL/HiL/Operation) Locate a bug	Visual inspection of logs Manual Debugging	Automatically reproduce a scenario in the laboratory using the information of the monitoring subsystem	Increase the number of bugs detected Effort saving in analysing problems in installations

level, continuously validating the system in an automated manner for all stages increases the number of bugs detected while reducing the time to prepare the validation infrastructure.

VI. Related Work

Microservice-based architectures are spreading in the Internet of Things (IoT) and CPS domains due to the high suitability of this paradigm for these fields, as they share some goals (e.g., lightweight communication, independent deployable software, etc.) [9].

Thramboulidis et al. [21] and Alam et al., applied microservice-based architectures to exploit its benefits in CPSs involved in industrial use cases. Specifically, Thramboulidis et al., [21] proposed a framework which uses model-driven engineering to semi automate the use of microservices on manufacturing systems, remarking the flexibility of such an architecture for plant processes. In [5], the authors combined Docker and microservices using a distributed and modular architecture to execute Industrial IoT (IIoT) applications, showing its validity for deployments on time-sensitive scenarios. These works propose developing microservice-based applications for CPSs, but do not use microservice-based solutions in the development process tasks.

As mentioned, the development of CPSs has typically suffered from long development life-cycles [4]. DevOps practices are now gaining attention in the CPS domain, and many works are focusing on applying different techniques such as Model-Driven Engineering [10] or Digital Twins [22] to ease and

enhance DevOps activities on CPSs. Many tools focus on specific life-cycle stages of CPSs, such as deployment [19] [11] [20], monitoring [17] [23] or validation [8] [15], but do not have whole life-cycle management capabilities, requiring the use of multiple tools to handle all life-cycle phases.

There are also some works which exploit the benefits of microservices to perform DevOps activities. [13] proposed applying the microservice design principles for software deployment and [16] presented a monitoring tool based on microservices, but these works focus on cloud infrastructures management, rather than CPSs.

VII. Conclusion and Future Work

In this work, we have instantiated part of the reference architecture presented in the H2020 Adeptness project for the Orona use case, and a prototype of the architecture for SiL validation has been developed.

Automation of DevOps activities has paramount relevance specially in CPSs, where the life-cycles are so long and the development tasks so fragmented, that easing and speeding these tasks may have a huge impact. For instance, automating software deployment can reduce maintenance effort in deploying the software in each device manually and continuous monitoring and validation may be helpful for understanding the system behaviour at run-time. esides, a microservice-based architecture offers high flexibility, simplifying the architecture's adaptation to different life-cycle stages, and allows scaling the solution to large-scale systems.

In the future, we plan to continue extending the architecture to support Orona's development activities in all life-cycle stages. We also plan on including additional mechanisms to ensure correct software deployment, such as applying Machine Learning techniques to detect performance problems in new software releases.

Acknowledgment

This publication is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319. This work has been partially supported by the Basque Government through the Elkartek program under the DIGITAL project (Grant agreement no.KK/2019-00095). Aitor Arrieta and Goiuria Sagardui are part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT1326-19), supported by the Department of Education, Universities and Research of the Basque Country.

References

[1] Adeptness project webpage: https://www.adeptness.eu/.

[2] Requirements-and-validation-tests – https://adeptness.eu/wp-content/uploads/2020/09/D1.1-ANNEX-A-Requirements-and-validation-tests.pdf.

[3] Requirements-and-validation-tests – https://adeptness.eu/wp-content/uploads/2020/11/D1.1-REQUIREMENTS$_v$1.1.*pdf*.

[4] Pekka Abrahamsson, Goetz Botterweck, Hadi Ghanbari, Martin Gilje Jaatun, Petri Kettunen, Tommi J. Mikkonen, Anila Mjeda, Jürgen Münch, Anh Nguyen Duc, Barbara Russo, and Xiaofeng Wang. Towards a Secure DevOps Approach for Cyber-Physical Systems. *International Journal of Systems and Software Security and Protection*, 11(2):38–57, 2020.

[5] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9):118–123, 2018.

[6] Jon Ayerdi, Aitor Garciandia, Aitor Arrieta, Wasif Afzal, Eduard Enoiu, Aitor Agirre, Goiuria Sagardui, Maite Arratibel, and Ola Sellin. Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 280–290. IEEE, 2020.

[7] Jon Ayerdi, Sergio Segura, Aitor Arrieta, Goiuria SagarduiMaite Arratibel, and Maite Arratibel. Qos-aware metamorphic testing: An elevation case study. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 104–114. IEEE, 2020.

[8] Sreram Balasubramaniyan, Seshadhri Srinivasan, Furio Buonopane, B. Subathra, Jüri Vain, and Srini Ramaswamy. Design and verification of Cyber-Physical Systems using TrueTime, evolutionary optimization and UPPAAL. *Microprocessors and Microsystems*, 42(2016):37–48, 2016.

[9] Bjorn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2016-November, 2016.

[10] Benoît Combemale and M. Wimmer. Towards a model-based devops for cyber-physical systems. In *DEVOPS*, 2019.

[11] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre Emmanuel Novac, Stephane Lavirotte, Jean Yves Tigli, and Arnor Solberg. GeneSIS: Continuous orchestration and deployment of smart IoT systems. In *Proceedings - International Computer Software and Applications Conference*, volume 1, pages 870–875. IEEE Computer Society, jul 2019.

[12] Javad Ghofrani and D. Lübke. Challenges of microservices architecture: A survey on the state of the practice. In *ZEUS*, 2018.

[13] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure DevOps. *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, pages 202–211, 2016.

[14] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[15] Elena Markoska and Sanja Lazarova-Molnar. Towards smart buildings performance testing as a service. *2018 3rd International Conference on Fog and Mobile Edge Computing, FMEC 2018*, pages 277–282, 2018.

[16] Marco Miglierina and Damian A. Tamburri. Towards omnia: A monitoring factory for quality-aware devops. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, page 145–150, New York, NY, USA, 2017. Association for Computing Machinery.

[17] K. Monisha and M. Rajasekhara Babu. *A novel framework for healthcare monitoring system through cyber-physical system*. Springer Singapore, 2019.

[18] R. O'Connor, Peter Elger, and Paul M. Clarke. Continuous software engineering—a microservices architecture perspective. *Journal of Software: Evolution and Process*, 29, 2017.

[19] Nenad Petrovic and Milorad Tosic. SMADA-Fog: Semantic model driven approach to deployment and adaptivity in fog computing. *Simulation Modelling Practice and Theory*, 2019.

[20] Luis F Rivera, Norha M Villegas, Gabriel Tamura, Miguel Jiménez, and Hausi A Müller. UML-driven Automated Software Deployment. *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 257–268, 2018.

[21] K. Thramboulidis, D. C. Vachtsevanou, and A. Solanos. Cyber-physical microservices: An iot-based framework for manufacturing systems. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 232–239, 2018.

[22] Miriam Ugarte Querejeta, Leire Etxeberria, and Goiuria Sagardui. Towards a devops approach in cyber physical production systems using digital twins. In *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*, pages 205–216. Springer International Publishing, 2020.

[23] Michael Vierhauser, Jane Cleland-Huang, Sean Bayley, Thomas Krismayer, Rick Rabiser, and Pau Grünbacher. Monitoring CPS at runtime - A case study in the UAV domain. *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018*, pages 73–80, 2018.