Future Generation Computer Systems 157 (2024) 360–375

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Enabling DevOps for Fog Applications in the Smart Manufacturing domain: A Model-Driven based Platform Engineering approach

Julen Cuadra [a,*], Ekaitz Hurtado [a], Isabel Sarachaga [a], Elisabet Estévez [b], Oskar Casquero [a], Aintzane Armentia [a]

[a] *Systems Engineering and Automatic Control Department, University of the Basque Country (UPV/EHU), Bilbao, 48013, Spain*
[b] *Electronics and Automation Engineering Department, University of Jaén, Jaén, 23071, Spain*

ARTICLE INFO

Keywords:
Fog Computing
Model Driven Engineering
Node-RED
Smart Manufacturing
DevOps
Platform Engineering

ABSTRACT

Cloud Computing is revolutionizing smart manufacturing by offering on-demand and scalable computer systems that facilitate plant data analysis and operational efficiency optimization. DevOps is a methodology, widely used for developing Cloud Computing systems, that streamlines software development by improving its integration, delivery, and deployment. Although cloud application designers within a DevOps team are assumed to have development and operational knowledge, this does not fall within the skills of experts that design analytics applications of plant data. The deployment environment is also relevant since, as such applications are often hosted in the Fog, the proliferation of application components may hinder their composition and validation. This work is aimed at embracing the Platform Engineering approach to provide a tailored toolkit that guides the design and development of OpenFog compliant applications for the experts in the Smart Manufacturing domain. The platform uses Model Driven Engineering techniques and a flow-based visual editor to allow application designers to graphically compose applications from components previously delivered by component developers, abstracting them from the underlying technologies. As a result, containerized applications, ready to be deployed and run by a container orchestrator, are obtained. The feasibility of the proposal is proved through an industrial case study.

1. Introduction

Cloud Computing has established itself as the most suitable solution to meet market demands in terms of data processing, storage, and optimization, which has led to the development of innovative applications related to Feature Engineering, data filtering or model training [1]. This is very promising for the Smart Manufacturing domain, since there is a great deal of interest in introducing artificial intelligence to make data-driven decisions that reduce error-prone human control [2], and provide valuable troubleshooting clues that guide experts in the rapid detection and resolution of faults [3]. Thus, Cloud Computing is seen as the default choice to check factory performance, adapt production to demand, and reduce production breakdowns due to defective products or equipment failures [4–6]. However, its adoption within existing factory automation remains a challenge: on the one hand, cloud applications must be integrated into the standard automation control hierarchy, while ensuring data security and time behavior requirements [7]; on the other hand, the engineering procedures used to design, develop and delivery cloud applications must be adapted to the heavily domain-oriented profile of the experts working in the automation field. As far as the authors know, this latter aspect has not yet been explicitly addressed in the literature.

Cloud Engineering is a field of engineering that applies systematic methods and support tools for conceiving, developing, operating, and maintaining cloud computing systems. As such, Cloud Engineering covers diverse aspects including automated management of applications run-time, architectural patterns for application design, and integration of application development and delivery procedures. However, Cloud Computing is still evolving, facing an increasing complexity in each of these aspects.

From the point of view of automated management of applications run-time, in certain domains, cloud engineers face the challenge of embracing the Cloud Continuum paradigm [8], which is usually structured in three levels: (1) the Edge, composed of physical assets providing services required in the different processes, and generating massive amounts of data; (2) the Fog, composed of heterogeneous and distributed devices called Fog Nodes, providing computing and storage

* Corresponding author.
E-mail addresses: julen.cuadra@ehu.eus (J. Cuadra), ekaitz.hurtado@ehu.eus (E. Hurtado), isabel.sarachaga@ehu.eus (I. Sarachaga), eestevez@ujaen.es (E. Estévez), oskar.casquero@ehu.eus (O. Casquero), aintzane.armentia@ehu.eus (A. Armentia).

https://doi.org/10.1016/j.future.2024.03.053
Received 22 November 2023; Received in revised form 27 March 2024; Accepted 31 March 2024
Available online 8 April 2024

resources located in an environment close to the source of the data; and (3) the Cloud, providing similar functionalities to the Fog, but with processing and storage resources located in external servers only accessible through the Internet [6,9–12]. This Cloud Continuum has a federated nature [7,13], as attested to by the fact that OpenFog, as a Fog Computing standard, states that "Fog Computing works with the Cloud" [14]. In the Smart Manufacturing domain, since plant assets generally lack the computing resources to support data analytics applications, these were initially deployed in the Cloud [1]. However, this approach led to undesired issues related to data security, networking performance and task conformance, so there is a current tendency to deploy such manufacturing applications in the Fog [7,11,15].

From the point of view of application design, applications in cloud environments have typically been designed as a composition of small and independent microservices, independently scalable and deployable [16]. The definition of microservices is a critical design challenge [17,18], as each component fulfills a specific functionality and requires an interface to interact with other components [19–21]. Microservices promote service reuse through individual Application Programming Interfaces (APIs) that implement simple functionalities. However, this creates reliability, performance and security challenges for the complex applications composed using those services or APIs [22]. Specifically, the proliferation of APIs creates the need for application designers to discover available microservices and determine the specific functionality associated with each service offered. Since each programmer may have their own criteria, standardization is necessary to compose these applications efficiently. Both in the Cloud and in the Fog, standardization efforts (such as OASIS Topology and Orchestration Specification for Cloud Applications -TOSCA- [23] and OpenFog [24], respectively) have emerged with the aim of establishing common rules, with synergies between the two standards: TOSCA defines applications as collections of services to define workloads and proposes to encapsulate them in CSARs (Cloud Service ARchives); similarly, OpenFog defines them as loosely coupled collections of microservices that are encapsulated into containers. Other authors, without leaning on said standards, go a step further and constrain applications as directed data flows (hereinafter workflow) that greatly simplify application logic [4,25–29]. Furthermore, considering that the use of Model Driven Engineering (MDE) techniques is suitable to guide and assist the design, development and deployment phases of complex systems [30], there have been efforts to adopt MDE techniques in Cloud Engineering [31].

From the point of view of application development and delivery procedures, it is common to use a DevOps model for streamlining and integrating the software development process with the delivery, deployment and operation of said software. Microservice development, application design and application operation skills are heavily domain-oriented aspects, and so is the composition of a DevOps team [32]. For instance, in the Smart Manufacturing domain, a Plant floor operator (a domain expert) may be interested in defining a fog application to perform quality control on a product or predictive maintenance on a machine. Plant floor operators know what plant data to read, what kind of processing is required and what kind of action is required, but they do not know how to specifically develop the application components, nor do they know how to deliver or deploy an application. Similarly, Production Managers (other domain experts) define production plans as a composition of the services offered by plant entities (machines, Mobile Manipulation Robots -MMR-, operators, etc.), but do not usually know how a Programmable Logic Controller (PLC) or MMR program is developed, nor how it is commissioned. In this sense, domain experts with the role of application designers must be abstracted from the technicalities of all the phases from developing application components to deploying applications. This is markedly different from traditional DevOps teams, where application designers are assumed to have both development and operational skills. Thus, the separation of concerns between the stakeholders that participate in a DevOps team is particularly important in the Smart Manufacturing domain. To this end, providing reusable and cooperative capabilities within a DevOps infrastructure becomes an enabling factor in a Cloud Engineering methodology. Platform Engineering is an emerging trend intended to provide an engineering platform specifically designed to support the needs of the different stakeholders by provisioning a common interface to a DevOps infrastructure [33].

In this context, the research goal of the present work is twofold, focused on covering the needs of the final product. On the one side, it proposes a generic MDE-based methodology that guides the design and development of OpenFog compliant applications for the main roles identified in the Smart Manufacturing domain. On the other hand, as any MDE-based methodology requires a support environment to simplify and automate the process, this work provides a platform that embeds the methodology in a toolkit tailored for containerized application component delivery; easy and validated application design as a composition of components; and their deployment in a container orchestrator.

The remainder of this paper is organized as follows. Section 2 analyzes how the literature addresses the inclusion of DevOps in the Cloud Computing paradigm and the design and development of containerized microservice-based applications. Section 3 is devoted to overview the proposed model-driven methodology and the implementation of the engineering platform that follows said methodology. Section 4 and Section 5 describe the platform as follows: while Section 4 details how to design, develop and deliver a Fog Component, Section 5 focuses on the design and development of a Fog Application. Section 6 presents a case study where the suitability and applicability of the platform is tested in a real case scenario in the Smart Manufacturing domain. Finally, Section 7 discusses a qualitative analysis and presents the conclusions obtained and the future work.

2. Related work

In 2018, researchers and practitioners published a manifesto [22] presenting the advancements of Cloud Computing and the challenges for achieving the Future Generation Cloud Computing in the following decade. Application Development and Delivery, and Cloud Computing at the Edge with Fog Computing were identified as open issues. In this context, model-based orchestration is increasingly being adopted in DevOps methodologies and within application delivery, automating lifecycle management and application configuration. Nevertheless, a shortage of application delivery frameworks and programming models is identified as an issue that research should undertake. The next paragraphs analyze several works that address these issues in some way.

Platform Engineering aims to support all the concerns of the stakeholders following a DevOps approach. Specifically, the application and extension of these concepts on the Cloud is coined as CloudOps in [8], where the authors extend the typical DevOps pipeline with the particularities that the continuum presents. This paradigm aims to solve the issues faced by Application Developers and Application Operators due to the challenges presented by this complex federated computational environment.

The authors in [34] present a complex automated pipeline generator with Continuous Integration and Continuous Delivery (CI/CD) principles for the deployment of multiple types of applications. This DevOps approach abstracts the stakeholders in the creation, management or monitoring of the pipelines, leaning on Docker [35] and Kubernetes [36] to achieve this. The DevOps pipeline is also leveraged in [37] for cyber physical systems, with the aim of automating the Continuous Delivery of operation ready software based on a microservice architecture that enables said automation. The same idea is followed in [38], where the digital twin is used as the main enabler of the DevOps approach in cyber physical production systems, serving as a bridge between the development and operation environments. Models are also used to ensure the fulfillment of application requirements, as proposed in [39], where the authors present an autonomous management framework for microservice-based Cloud-native applications, in accordance with a Model Driven Architecture (MDA). The models presented in the paper are developed based on technology-agnostic meta-models that follow a DevOps approach.

Regarding application design and development, the work in [25] proposes the Distributed DataFlow (DDF) programming model for developing IoT (Internet of Things) applications that can be deployed on Fog and Cloud devices. The applications are represented as directed graphs where the nodes of the graph are independent processing units, which have inputs and outputs. The authors distinguish two stakeholders: node developers and application developers. While for the former no development model is indicated, for the latter, an extension of the Node-RED [40] tool is proposed, allowing the graphical design of workflows. Similarly, the authors in [41] also focus on DDF-based IoT applications. Depending on real-time requirements of the component, it is possible to benefit of the advantages of the Edge devices or the resources offered by the Cloud. The authors develop their own graphical interface that allows the design of the network topology and the application graphs. The authors in [29] also define meta-models for the design of applications as workflows of reactive processing tasks. Although meta-models are used by the architecture to automate application deployment, this work lacks graphical design tool for applications.

In this context, some authors make use of modeling techniques to ease the application design and/or development, as in [42], where a Domain Specific Language (DSL) is proposed to enable the design of multi-tier applications, covering IoT devices as well as Edge, Fog and Cloud nodes. This DSL is collected in two meta-models: one devoted to the definition of IoT systems and another focused on the definition of adaptation rules that allow the system to automatically respond to different situations. This work also provides a Model-to-Text (M2T) transformer to generate the necessary application deployment files. Another type of applications are used in [43], where an Unified Modeling Language (UML) DSL is proposed to support the design and continuous deployment of applications, leveraging TOSCA for their deployment. Three meta-models are presented following the MDA approach: a platform-independent model to assist developers, a technology-specific model for evaluating the architecture and a deployment-specific model to automate the deployment of the application in the cloud.

TOSCA is also used in [44], as well as Docker, where the deployment of multi-component applications on top of existing container orchestrators is enabled. The proposal achieves component-aware management, as they decouple the lifecycle of application components from that of the containers hosting them, but the application designer must have a detailed knowledge of the technologies used, the modeling of applications and the development of the necessary software components. There is no separation of concerns between programmers and designers and practitioners in the industry usually prefer to use more informal approaches for describing microservice based applications [17]. A similar approach is proposed in [45], where a model-driven cloud application orchestration approach that enables a role-aware orchestration is proposed, transforming TOSCA-based Cloud applications into Open Application Model (OAM) [46] files ready to be deployed in Kubernetes. This approach is centered around the abstraction of platform specific knowledge from the application developers and to enable a separation of concerns between them and application operators following typical DevOps practices.

Usually, the application concept is not considered from the management point of view, which is addressed in [47], where the authors propose an extension of Kubernetes platform, called Application-Centric Orchestration Architecture (ACOA), with the aim of achieving an application deployment on the Edge-Fog-Cloud continuum that maximizes the quality of service of the deployed application. For this purpose, an application model based on directed graphs is proposed, where components are the vertices, and the channels are the edges. The application is considered as the smallest deployment element, and, although components are deployed separately, they are related when the application is composed. Similarly, application-aware orchestration is treated in [48], where the authors propose an OpenFog-compliant Kubernetes extension that enables the inclusion of the Application and Component concepts at the orchestration level. These concepts are managed by custom controllers that operate said resources as Kubernetes native resources. Furthermore, the authors propose a Hierarchical Application Management Structure to adapt the proposal to any domain and operating model of an organization or system.

Container-based applications in Fog Computing are also addressed in [49], where the Semantic Model driven Approach to Deployment and Adaptivity of these applications (SMADA-Fog) is presented. This approach contributes modeling tools or a semantic framework in relation to the deployment of the applications. However, although a deployment meta-model is presented, a meta-model for applications is not proposed. The semantic framework enables the automatic processing of complex knowledge related to automated code generation for both service deployment and adaptivity.

On the other hand, regarding the development process Kubernetes and Docker technologies are also use as edge infrastructure. This is the case of SODALITE@RT [50], an open-source framework capable of modeling cloud–edge microservice-based IoT applications. In this work TOSCA is adopted to describe the deployment model of a managed heterogeneous distributed application and automate their deployment, monitoring, or the management. Furthermore, in Edge Computing there are also other frameworks for IoT applications, such as EdgeFlow [51], capable of assisting the developer in the application development process, dividing applications functionality into multiple parts, defining and validating the requirements and finding a deployment strategy. IoT application deployment plans are also addressed in [52], describing a model-based approach to automatically assigning multiple software deployment plans to hundreds of edge gateways and connected IoT devices. A platform-independent meta-model describes a list of target devices and deployment plans and is validated with a prototype integrated into a DevOps toolchain.

As far as authors know, there is not a proposal of an engineering platform that focuses on the design and development of container-based components that are later used to build applications as a composition of microservices. Furthermore, DevOps requires the quick and easy delivery and deployment of the applications which can be achieved through the use of a graphical tool that is part of the toolkit implementing the platform. The operational aspect of the pipeline is widely covered, but the development aspect is implemented through ad-hoc solutions that lack the reusability and inter-domain operability required today.

3. Overview of the approach

As stated above, the separation of concerns between the different stakeholders that make up a DevOps team is crucial. As the specific needs of a DevOps team vary from domain to domain, so does the engineering platform, which, as a product, depends entirely on the needs of its end users. As a motivating scenario, Fig. 1 presents the DevOps pipeline, customized for the Smart Manufacturing domain applications, shown as a feedback loop to describe the relationship between the development and operation tasks.

Against this background, this section is divided into two subsections. The first subsection presents a methodology based on MDE to guide the design, development, delivery, and operation of Fog-level applications, considering them as microservice workflows that are implemented as containerized components. The second subsection presents the platform that implements said methodology, focusing on the Dev aspect and delegating the Ops side on the leading container orchestrator.

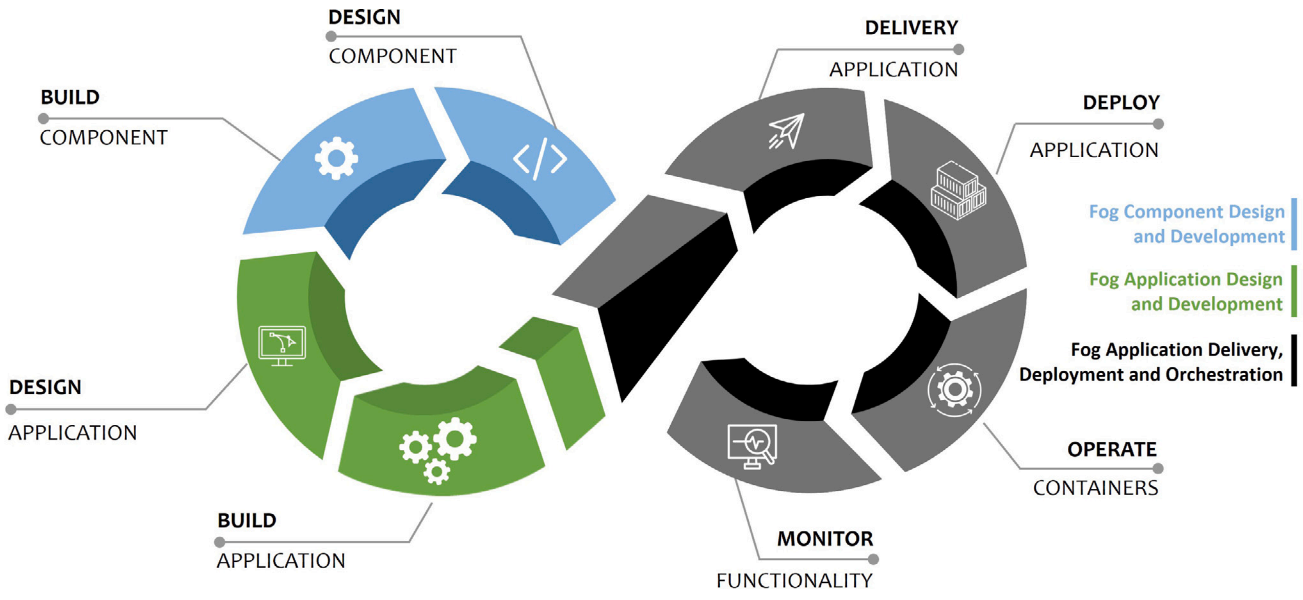


Fig. 1. Custom DevOps pipeline for the entire Fog application development and operation process.

3.1. Model-driven methodology for the design, development, delivery, and operation of Fog Applications

The proposed methodology is based on MDE techniques and is aimed at decoupling the different phases of the Fog Application lifecycle, from the component and application design and development to the application delivery, deployment, and operation. The proposal considers three stakeholders: Component Programmers, Application Designers and Platform Administrators (whose tasks are marked in Fig. 2 in blue, green, and black colors, respectively). In order to achieve the separation of concerns between them, this work also defines two concepts, the so-called Fog Components and Fog Applications, which are based on the Component Based Software Engineering (CBSE) approach [53]. CBSE proposes the construction of complex systems (Fog Applications) by means of the composition of simple components (Fog Components), which are previously developed independently of the applications. The CBSE approach promotes the development of components as software modules that can be reused in different applications. Thus, in the present work, Component Programmers are in charge of designing Component Models and developing Fog Components (software modules that encapsulate several functionalities) before application design. Component Programmers store them in a Fog Component Catalog. This element achieves the interoperability between Component Programmers and Application Designers, allowing Application Designers to conceive Fog Components as black boxes that offer functionalities and use them when necessary.

To comply with the OpenFog reference architecture, this paper proposes that Fog Applications are composed of instances of Fog Components for which just one of their functionalities has been selected for a given application. Therefore, the Fog Component instance will correspond to a microservice, and the selected functionality will be its offered service. This way, reusability is achieved as the same Fog Component can be used in different Fog Applications as different microservices. To this end, OpenFog proposes to encapsulate microservices using containerization technologies. The Application Designers are in charge of generating the Application Model, so they have to instantiate, customize and connect each of the desired components from the catalog.

Therefore, as it is depicted in Fig. 3, in the early stages of the development of a Fog Component the Component Programmer must develop its source code considering all the functionalities it might offer in any application. These may be functionalities that are related in some way or that the programmer has decided to package together. Besides, they will identify the software dependencies that the Fog Component needs, encapsulating all the pieces in a container image (Requirements of Fog Component in Fig. 3). Fog Component design comprises building the component model that describes such Fog Component implementation, which is needed by the Programmer to generate a new entry in the Fog Component Catalog (Catalog item of Fog Component in Fig. 3). These items are ready to use by the Application Designer but lack the context to be part of an application. For this purpose, they make use of M2T transformations (ModelToComponent transformation rules in Fig. 2).

The Application Designer will instantiate the Fog Components providing them with context within the Fog Application. This implies determining the required components from the Fog Component Catalog and selecting one their available functionalities. When these items are instantiated in the flow-based visual editor, and parameterized as part of an application, they are turned into microservices (ready-to-deploy Microservice in Fig. 3). This element is aware of its “surroundings” (previous and following components in the flow) and “responsibilities” (service to execute) within the context of the application. When the Platform Administrator decides to deploy the application, the microservices will be orchestrated as containers on the container orchestration platform (Runtime Microservice in Fig. 3), executing the desired service (functionality selected from the Fog Component) according to its parametrization.

It should be noted that the Application Models generated by Application Designers are platform-agnostic files, independent of each orchestrator’s deployment intricacies. To deliver and operate Fog Applications, this generic Application Model is transformed into the necessary ready-to-deploy file(s), named as the Application Delivery Model. The Platform Administrators are in charge of this transformation, as they are the ones who deploy the application, being experts in the selected container orchestration platform. This platform is responsible of deploying and operating the Application Delivery Model, running the individual microservices, enabling their communication and checking whether the application is correctly running.

3.2. Supporting technologies for the implementation of the engineering platform

In order for the engineering platform to implement the proposed methodology, the selection of the supporting technologies is critical as there are hundreds of options to choose from, each with their

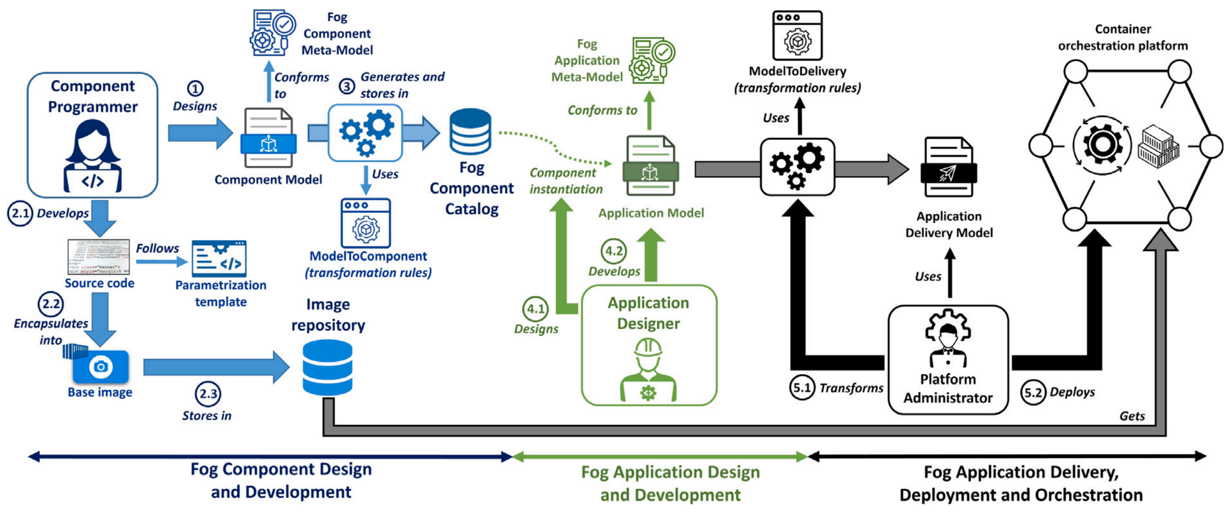


Fig. 2. Overview of the model-driven methodology for the different phases of the development lifecycle of Fog Applications.

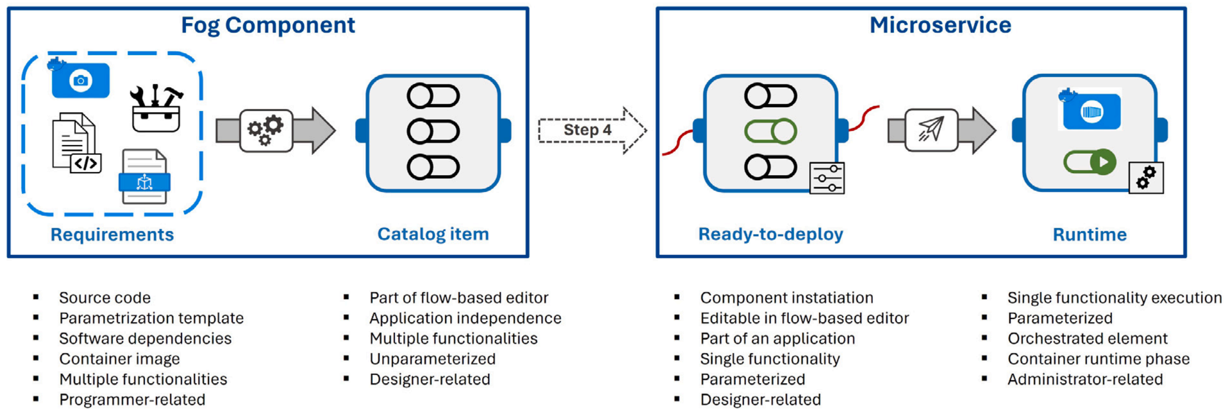


Fig. 3. Relation between the Fog Component and Microservice concepts.

benefits and disadvantages, as showcased by the Cloud Native Computing Foundation's landscape [54]. The chosen technologies must offer enough tools to automate the process and guarantee the validity of the results. Furthermore, they must provide aspects such as simplicity or flexibility and allow for the desired separation of concerns between the stakeholders. Thus, the authors have performed an analysis of different options for each technology integrated in the process. Fig. 4 shows the engineering platform that implements the model-driven methodology, highlighting the supporting technologies involved.

This work proposes the use of XML (eXtensible Markup Language) technologies throughout the entire process to ensure the validity of the generated components and applications. Models are developed as XML documents (.xml file extension) whereas the authors selected W3C XML Schema Technology [55] (.xsd file extension) for the implementation of the meta-models. For M2T transformations the authors propose the use of the XSLT technology stylesheets [56], which allows transforming XML documents to documents in other formats.

On the one hand, containers have become the de facto standard for packaging microservices, because they allow for light-weight virtualization at the operating system level [20,57], in concrete, Docker [35] has been selected as it is the leading tool for creating containers [44, 58,59]. Fog Component base images are Docker container images, and they are stored in the Image Repository. Docker offers some features suited to implement the proposed methodology, i.e., reusability, cost-effectiveness or giving freedom to Component Programmers to develop the source code with any programming language. Furthermore, the use of containers enables the separation of concerns between Component Programmers, Application Designers and Platform Administrators [60].

On top of that, with the objective of avoiding manual tasks and easing the design and development of Fog Applications, this work proposes the use of a flow-based visual editor implemented as an extension of the Node-RED tool. Node-RED is a visual programming tool with graphic potential that allows a visual workflow definition, based on relating different functionalities represented as nodes. Another potential of Node-RED lays in its capacity to develop customized nodes [26] from user defined templates [61]. This work proposes leveraging this fact, by integrating the Fog Component Catalog as a Fog Computing Library within Node-RED, so that Fog Components are stored as nodes. The Fog Computing Library aims to achieve the desired complete separation of concerns. Besides, in Node-RED, these library nodes can execute personalized tasks. More precisely, the authors contribute the customization of Node-RED using scripts that are also automatically generated through M2T transformations applied to component models, to generate nodes that can be used and parameterized by the Application Designer and are able to generate the Application Model automatically. With all these features, the proposed modified Node-RED can offer a graphic tool to design and develop the Application Model automatically.

On the other hand, application development is a process tightly coupled with the orchestration platform chosen to manage the deployment and lifecycle of containers [62]. For the platform to be realizable, the container orchestration platform chosen must: (1) enable the creation of containers and manage their lifecycle, (2) obtain the information packaged in the Application Delivery model and pass it to the individual containers as environment variables and (3) enable

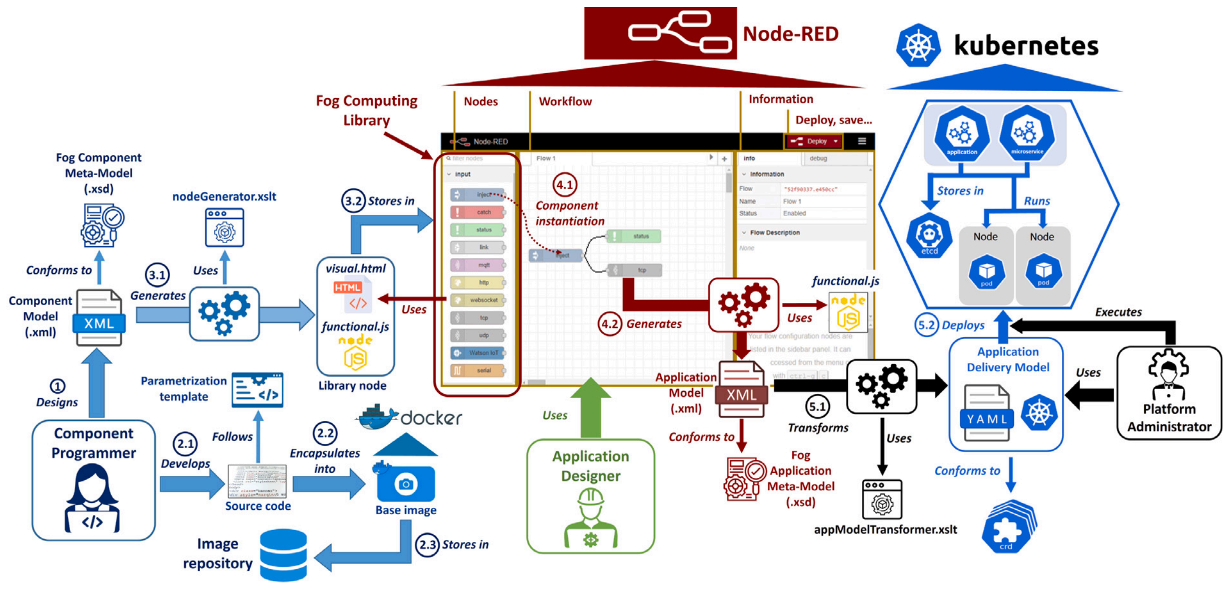


Fig. 4. Engineering platform that implements the model-driven methodology for the different phases of the development lifecycle of Fog Applications, displaying the technologies involved.

the communication between microservices. In this work, the authors propose to delegate the operation of Fog Applications in Kubernetes, as it is the most popular platform for the orchestration of container-based applications [58,59], and fulfills all of the aforementioned requirements. Furthermore, Kubernetes offers some functionalities that aid in the delivery and operation of Fog Applications, based on containerized microservices, such as application scalability, workload optimization or self-healing capabilities for deployed containers. The platform-agnostic Application Model is transformed to the deployable Application Delivery Model through M2T transformations and XSLT technology.

4. Design and development of Fog Components

This work proposes a generic (domain independent) and abstract (application independent) Fog Component specification, based on a *Fog Component meta-model*, which gathers the minimum characteristics of a Fog Component. The design and development of Fog Components involves three steps, described in the following subsections (from Step 1 to Step 3).

4.1. Step 1: Design of a Fog Component

For each Fog Component, the Component Programmer must create a *component model* (.xml file) where they establish the functionality or functionalities that the Fog Component will encapsulate. For this, the Component Programmer follows the rules established by the proposed Fog Component meta-model (see Fig. 5), which is implemented as an XML Schema (.xsd file).

Each *component* is characterized by its *name* as a unique identifier in the system and a more detailed *description* can be provided. To ease application design, Fog Components are grouped according to the *category* attribute. The number and variety of categories are domain-dependent (*TComponent* type) and established by the Component Programmer. Finally, information related to the component implementation is provided (*imgBase* attribute).

Each component can offer one or more functionalities (elements of the *Function* type), defined by a unique identifier (*id* attribute), its *name*, and the set of input and/or output parameters needed by each of the functionalities (*inputs* and *outputs*, respectively). It is also possible to provide a more detailed description of the functionalities and to provide additional parameters related to the component runtime through the *customization* attribute. The input and output parameters are defined as a data structure characterized by its name and data type (*dataType* attribute). The required data types are determined by the Component Programmer when defining the interface of the functionalities. Finally, the communication *protocol* supported by each functionality must be specified, which may vary for each functionality or component.

As an example, Fig. 6 shows the component model related to the *AssemblyStation* Fog Component. It belongs to a category called *processing*, and it offers three different functionalities, named: *Calculate OEE*, *Calculate Performance* and *Calculate Trend*, each with their corresponding identifiers. In this example, every functionality receives a unique data structure with the required parameters to run properly, generating, as a result, another data structure with the output parameters, using the Hypertext Transfer Protocol (HTTP) protocol.

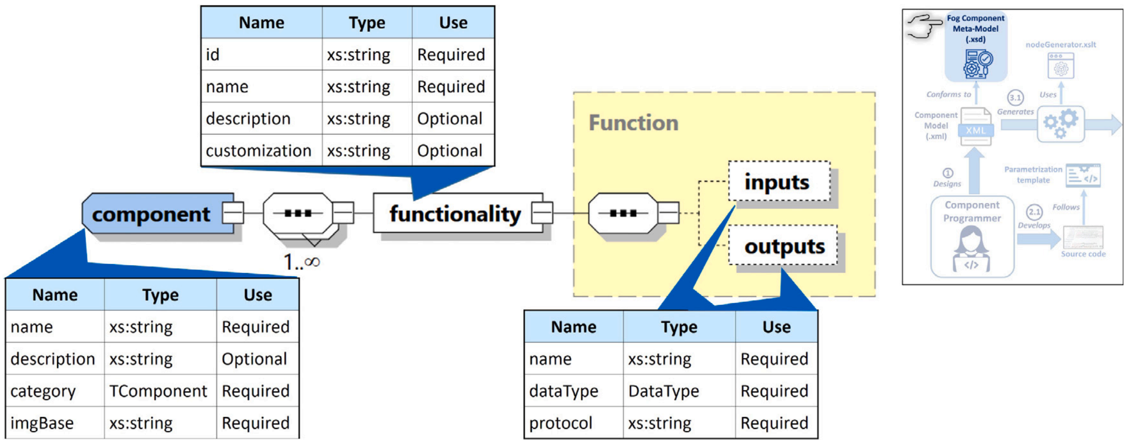
Finally, The Component Programmer checks whether the component model conforms to the Fog Component meta-model. The use of models, meta-models and an XML parser, ensures that the components designed are valid so they can be properly developed.

4.2. Step 2: Develop a Fog Component

Fog Component development is a process tightly coupled with the virtualization technology chosen. The Component Programmer must develop the *source code* that is able to execute the functionality or functionalities indicated with the established inputs and/or outputs and encapsulate it in a Docker image.

4.2.1. Step 2.1: Develop the Fog Component's source code

In this context, this work proposes that each Fog Component developed is packaged into a container image capable of running all of the component's functionalities, herein referred to as *base image*. It must be mentioned that, with the purpose of reusing the same base image to create different microservices that belong to different applications, it is mandatory to prepare base images to be instantiable in a customized manner within an application. Accordingly, the component code packaged in the container image must be made parametrizable with environment variables (*parametrization template*). These variables are categorized in two groups:

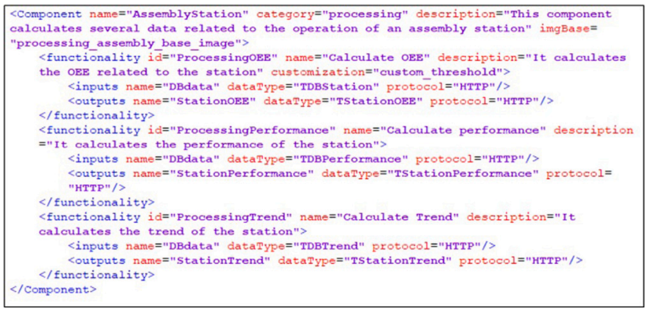


Name	Type	Use
id	xs:string	Required
name	xs:string	Required
description	xs:string	Optional
customization	xs:string	Optional

Name	Type	Use
name	xs:string	Required
description	xs:string	Optional
category	TComponent	Required
imgBase	xs:string	Required

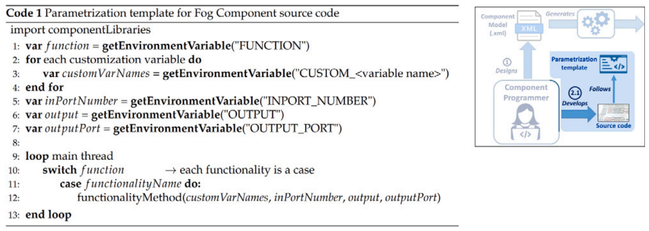
Name	Type	Use
name	xs:string	Required
dataType	DataType	Required
protocol	xs:string	Required

Fig. 5. Fog Component meta-model provided as an XML Schema.



```xml
<Component name="AssemblyStation" category="processing" description="This component
calculates several data related to the operation of an assembly station" imgBase=
"processing_assembly_base_image">
    <functionality id="ProcessingOEE" name="Calculate OEE" description="It calculates
    the OEE related to the station" customization="custom_threshold">
        <inputs name="DBdata" dataType="TDBStation" protocol="HTTP"/>
        <outputs name="StationOEE" dataType="TStationOEE" protocol="HTTP"/>
    </functionality>
    <functionality id="ProcessingPerformance" name="Calculate performance" description
    ="It calculates the performance of the station">
        <inputs name="DBdata" dataType="TDBPerformance" protocol="HTTP"/>
        <outputs name="StationPerformance" dataType="TStationPerformance" protocol=
        "HTTP"/>
    </functionality>
    <functionality id="ProcessingTrend" name="Calculate Trend" description="It
    calculates the trend of the station">
        <inputs name="DBdata" dataType="TDBTrend" protocol="HTTP"/>
        <outputs name="StationTrend" dataType="TStationTrend" protocol="HTTP"/>
    </functionality>
</Component>
```

Fig. 6. XML file related to the component model of the AssemblyStation Fog Component.



Code 1 Parametrization template for Fog Component source code

```
import componentLibraries
var $function$ = getEnvironmentVariable("FUNCTION")
for each customization variable do
    var $customVarNames$ = getEnvironmentVariable("CUSTOM_<variable name>")
end for
var $inPortNumber$ = getEnvironmentVariable("INPORT_NUMBER")
var $output$ = getEnvironmentVariable("OUTPUT")
var $outputPort$ = getEnvironmentVariable("OUTPUT_PORT")

loop main thread
    switch $function$            → each functionality is a case
        case $functionalityName$ do:
            functionalityMethod($customVarNames, inPortNumber, output, outputPort$)
end loop
```

Fig. 7. Parametrization template related to the implementation of the functionalities of the components shown as generic pseudocode.

- Environment variables that refer to the component instantiation. These environment variables are used to indicate: the functionality (*function* variable) selected to be executed when the component is instantiated within a Fog Application (microservice), and the variables related to the execution of the functionality selected (*customization* variables). These variables, always named as *CUSTOM_<variable name>*, are set by the programmer, and they are used to provide additional functionality-specific parameters to the container runtime.
- Environment variables that refer to the workflow of microservices in the Fog Application. Here, the fact that microservices (instances of Fog Components) communicate with other microservices is considered. If the functionality selected needs an input, the *INPORT_NUMBER* environment variable specifies the port number where the microservice will be listening. In case of having functionalities that produce output data, two environment variables are needed: *OUTPUT* to determine the next microservice name and *OUTPUT_PORT* to know the port on which it will be listening.

Note that the source code can be built in any programming language, since, by using Docker, the image will be functional. As an example, Fig. 7 shows the parametrization template that the Component Programmer must follow, described as generic pseudocode. As observed, the code has been constrained in terms of the environment variables described above. The concrete values of these variables are set during the application design phase.

4.2.2. Steps 2.2 and 2.3: Encapsulation and storage of the base image of a Fog Component

When the programmer writes the source code, they encapsulate it in a Docker image. This constitutes the base image of the microservice, which has all the functionalities programmed. Fig. 8 shows the Dockerfile template to generate the base image of a component. In the *FROM* command, the programmer must add the official image of the programming language selected to develop the source code. Using the *RUN* command all the necessary dependencies can be installed and the *COPY* command allows all required files, such as the source code file, to be included in the containing image. Eventually, in the *CMD* command the programmer must write the command to execute the source code file.

Using the aforementioned Dockerfile, the Component Programmer generates the base image of the component (Step 2.2) and, after that, stores it in a repository (image repository) for its later use (Step 2.3). The name of the image is obtained from the imgBase attribute of the component model.

4.3. Step 3: Integration of a Fog Component in the Fog Computing Library

As previously stated, this engineering platform pursues the separation of concerns between the different stakeholders that take part throughout the lifecycle of Fog Applications. The Fog Component Catalog (or Fog Computing Library in Node-RED) aims to decouple the tasks of the Component Programmer from those relative to the Application Designer. The component designed and developed must be integrated in the Fog Computing Library for its posterior use in the design of Fog Applications (Step 3), as a library node which is composed of two files, involving the transformation of the component model (Step 3.1) and the delivery of the resulting node in the library (Step 3.2). To automatically generate these files, the Component Programmer is provided with one XSLT stylesheet for each.

```
FROM programming_language_base_image

RUN install required dependencies

COPY sourceCodeFile pathInsideCointainer

CMD run sourceCodeFile command
```

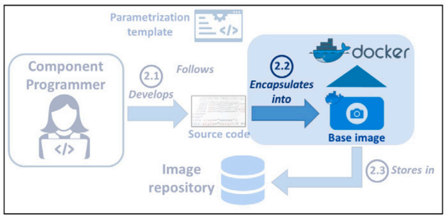


Fig. 8. Dockerfile for the generation of the base image of Fog Components.

For the automatic generation of the Fog Computing Library, it must be considered that Node-RED's runtime is built on Node.js. In fact, the library nodes are Node.js in which own JavaScript code may be written. On the one hand, to graphically design Fog Applications, the Application Designer must be aware of the Fog Components stored in the Fog Computing Library, and these must have a graphical representation. On the other hand, part of the information needed for application development is contained in the component models, and it depends on the concrete component instantiations. Therefore, this work proposes that each library node consists of two parts: (1) a visual part that allows displaying the library node content (*visual.html*) and (2) a functional part that automates the generation of the application model (*functional.js*). Thus, the M2T transformations in charge of creating the library node from the XML component model will use two XSLT stylesheets to generate these two files resulting in a node which can be directly integrated in the Fog Computing Library.

5. Design and development of Fog Applications

Once the Fog Components are designed (and stored in the Fog Computing Library) and developed (and stored in the image repository), the design and development of the Fog Applications becomes possible. The use of a meta-model is key, as the proposed *Fog Application meta-model* provides the structure required by the Application Designer.

5.1. Step 4.1: Design of a Fog Application

For each Fog Application, the Application Designer must obtain an application model, determining the microservices that compose it and establishing relationships between them, in order to arrange the Fog Application workflow. The application model must follow the rules defined by the proposed *Fog Application meta-model* (see Fig. 9), which is also implemented as an XML schema (.xsd file). In Fig. 9, relationships between elements of the meta-model are graphically represented. In turn, relationships between attributes are represented in textual form (gray notes associated with the elements).

This meta-model establishes that each *application*, characterized by its *name* as a unique identifier in the system, is composed of a set of two or more *microservices*. Each microservice corresponds to the instance of one of the Fog Components designed as described in Section 4. Therefore, the microservice *name* is inherited from the name of the instantiated Fog Component. The *service* offered by the microservice is inherited from the identifier of the selected function. The base image and the custom parameters for the function runtime are inherited by the *imgBase* and *customization* parameters from the equally named component model parameters.

Both the selected Fog Component and the selected functionality determine the communication needs of the microservice, which may have an input port (*inPort* element) and/or an output port (*outPort* element). Input and output ports share some attributes. Namely, their *name* as a unique identifier in the system, the supported *protocol*, and the type of the data structure required by the service (*dataType* attribute). The latter two are inherited from the selected functionality. The input port is also characterized by the port *number* on which the service will be listening for requests.

The data flow between microservices is defined by the communication *channels* established between ports. The channel starts in an output port of a microservice (*from* attribute) and finishes in an input port of another microservice (*to* attribute). Thus, the start and the end of the application data flow is determined by those microservices that have only an output port and only an input port, respectively. For the established channel to be considered valid, two conditions must be met: (1) the protocol supported by the two ends must be the same; (2) the data structures associated with the communicating ports must be of the same type.

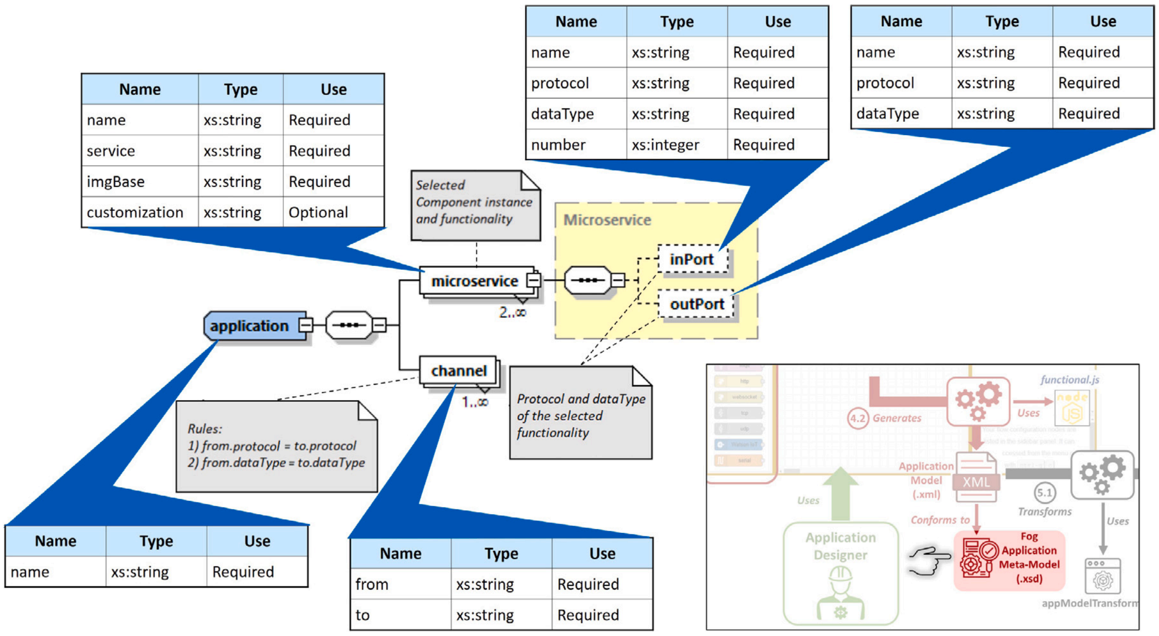
As an example, Fig. 10 presents the model related to a Fog Application composed of four microservices. There is an initial eXist microservice (it does not have input port) which sends information to a middle *AssemblyStation* microservice (it has input and output ports). Another intermediate microservice, *OEEEvents*, receives the result on its input port and, after performing its functionality, sends the data to the *Influx* microservice through its output port, which has no output port, ending the workflow. All microservices communicate through the HTTP protocol.

The graphic interface of the platform allows Application Designers to design Fog Applications as a workflow of microservices (Step 4.1). Node-RED offers a web view where microservices can be visually arranged as a workflow, which is detailed in Fig. 11. As mentioned in Section 4.3, nodes are composed of two parts: a visual part (*visual.html*) and a functional part (*functional.js*). The former is used in this web view, and it provides tools for Application Designers to customize the component and connect it to other instantiated components in order to design the application. Therefore, the Application Designer must select the desired Fog Component from those available on the Fog Computing Library, and after that, select the functionality desired for the current application (component instantiation). Furthermore, in case of the component has input, the Application Designer must indicate the port number where the corresponding service will listen and parameterize any custom variable if needed. Fig. 11 shows the graphical representation of the design and development process for the application model depicted in Fig. 10. Each microservice is represented by a rectangle with the name of the selected function (the service it offers). Channels are represented by lines that join microservices. The node edition part of this web view details how the functionality is selected and where to establish the input port number.

5.2. Step 4.2: Develop a Fog application

When all the needed Fog Components have been instantiated, the Application Designer requests the development of the application, that is, the generation of the Application Model, which the proposed Node-RED-based platform performs automatically. The second file that conforms the component's library node (its functional part or the so-called *functional.js*) runs when the Application Designer requests the application development and has the necessary functionalities to generate the Application Model.

The automatic generation of the Application Model is performed by a cascade process, illustrated in Fig. 12, as each node adds its information to the XML model. There are three types of nodes: the "initial" node (without input) creates the XML model; if the node



Name	Type	Use
name	xs:string	Required
service	xs:string	Required
imgBase	xs:string	Required
customization	xs:string	Optional

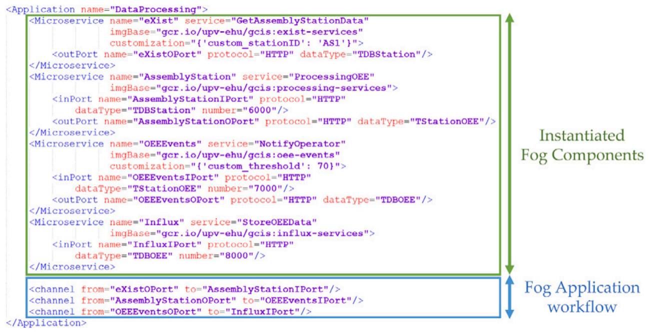
Name	Type	Use
name	xs:string	Required
protocol	xs:string	Required
dataType	xs:string	Required
number	xs:integer	Required

Name	Type	Use
name	xs:string	Required
protocol	xs:string	Required
dataType	xs:string	Required

Name	Type	Use
name	xs:string	Required

Name	Type	Use
from	xs:string	Required
to	xs:string	Required

Fig. 9. Fog Applications meta-model implemented as an XML Schema.



```xml
<Application name="DataProcessing">
    <Microservice name="eXist" service="GetAssemblyStationData"
                imgBase="gcr.io/upv-ehu/gcis:exist-services"
                customization="{'custom_stationID': 'AS1'}">
        <outPort name="eXistOPort" protocol="HTTP" dataType="TDBStation"/>
    </Microservice>
    <Microservice name="AssemblyStation" service="ProcessingOEE"
                imgBase="gcr.io/upv-ehu/gcis:processing-services">
        <inPort name="AssemblyStationIPort" protocol="HTTP"
            dataType="TDBStation" number="6000"/>
        <outPort name="AssemblyStationOPort" protocol="HTTP" dataType="TStationOEE"/>
    </Microservice>
    <Microservice name="OEEEvents" service="NotifyOperator"
                imgBase="gcr.io/upv-ehu/gcis:oee-events"
                customization="{'custom_threshold': 70}">
        <inPort name="OEEEventsIPort" protocol="HTTP"
            dataType="TStationOEE" number="7000"/>
        <outPort name="OEEEventsOPort" protocol="HTTP" dataType="TDBOEE"/>
    </Microservice>
    <Microservice name="Influx" service="StoreOEEData"
                imgBase="gcr.io/upv-ehu/gcis:influx-services">
        <inPort name="InfluxIPort" protocol="HTTP"
            dataType="TDBOEE" number="8000"/>
    </Microservice>
    <channel from="eXistOPort" to="AssemblyStationIPort"/>
    <channel from="AssemblyStationOPort" to="OEEEventsIPort"/>
    <channel from="OEEEventsOPort" to="InfluxIPort"/>
</Application>
```

Fig. 10. XML document related to the application model of the DataProcessing Fog Application.

is in the middle of the workflow (has input and output), it updates the model delivered by the previous node, and sends it to the next node; and the "final" node (without output) also updates the received model, but additionally executes the validation of the model against the application meta-model (see Fig. 9). If the designed application is correct, it will display the XML Application Model; if it is not, a warning message is issued, notifying the Application Designer which component is wrongfully defined and what the validation error is.

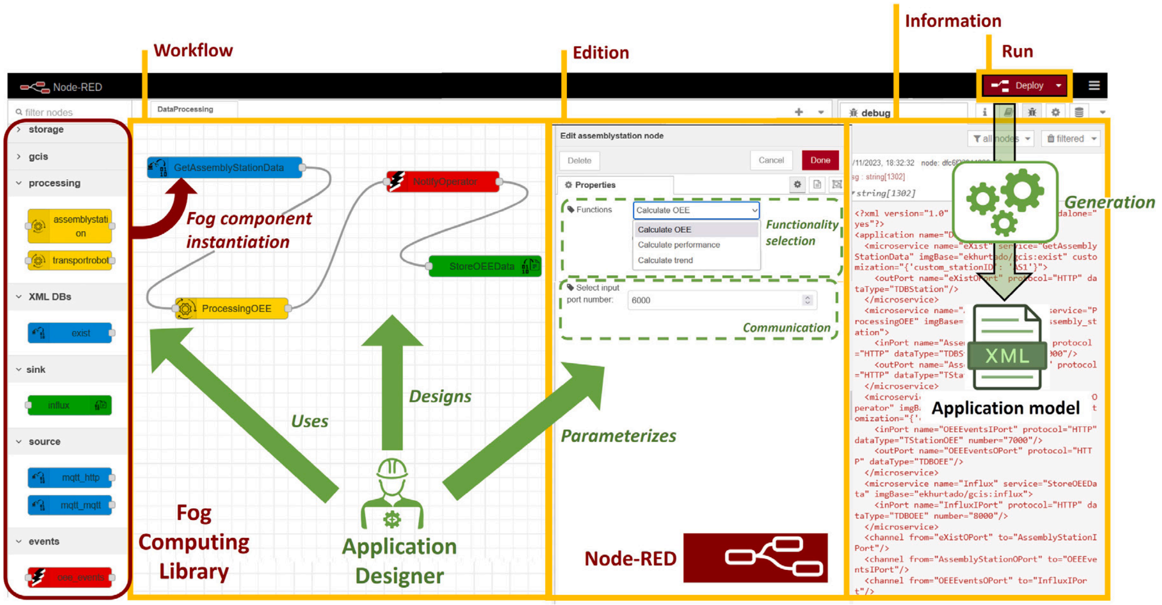
This automation, together with the use of the Fog Computing Library, abstracts the Application Designer from the complexity of the components and the model generation. The graphical potential of the tool offers a simple and efficient toolkit as the application model is always validated to ensure that it complies with the rules established by the Fog Application meta-model.

The next step, which is out of the scope of the proposed platform consists of building the Application Delivery Model (file/s to be deployed on the container orchestration platform chosen). However, the proposed methodology establishes that the Application Delivery Model can be automatically created from the corresponding Application Model (generic and platform-agnostic XML file generated by Node-RED). More precisely, this work proposes that the Platform Administrator generates it by means of M2T transformations applied to the Application Model. The required transformations are implemented in an XSLT stylesheet (*appModelTransformer.xslt* in Fig. 4). It should be noted that this stylesheet may vary as it is dependent both on the container orchestrator of choice and on the Platform Administrator. In fact, the orchestrator may offer different ways of inputting the Application Delivery Model and the Administrator is the one who makes the decision of which one to use. In this sense, this model allows the addition of specific metadata to customize the orchestration. Thus, it would be possible to include orchestration metadata related to scalability or network metrics for quality-of-service management [7,47] in the Application Delivery Model.

6. Case study

One of the main pillars of Industry 4.0 is the ability to turn the data collected from the plant into valuable information to improve the efficiency and productivity of the plant. To that end, it is necessary to ensure access, storage, and processing of the data. The Fog Computing paradigm suits these requirements thanks to its low latencies and data security. This section exemplifies how the proposed platform can be used within the Smart Manufacturing domain and its applications, testing the suitability and applicability of the platform engineering approach to a real case scenario. The manufacturing system used in this case study is structured in two levels (see Fig. 13): plant (which corresponds with the Edge) and Fog. A video is provided as Supplementary Material to ease the understanding of the usage of the proposed platform, as well as the code as a GitHub repository.

At the plant level, the case study has two robotic assembly stations (*AS1* and *AS2*), interconnected by a transport robot (*TR1*). Stations AS1 and AS2 have been designed to perform the assembly of a set of 3D printed parts emulating a stepper motor shaft. Each resource will publish different types of information via the Message Queuing Telemetry Transport (MQTT) [63] communication protocol (recommended by OpenFog), which decouples the communication of the two layers in an asynchronous manner. At the Fog level, two applications are proposed for each plant resource (AS1, AS2 and TR1): one for data acquisition and another for data processing. For simplicity, Fig. 13 and the rest of the section only show the applications related to a single plant resource (AS1), where blocks represent microservices, but with minimal changes, it can be extended to the AS2 resource, as only the resource ID in the submitted data structures needs to be changed.



```
/11/2023, 18:32:32  node: dfc6f
sg : string[1302]
▾string[1302]
<?xml version="1.0"                    dalone="
yes"?>
<application name="D
  <microservice name="eXist" service="GetAssembly
StationData" imgBase="ekhurtado/gcis:exist" custo
mization="{'custom_stationID': 'AS1'}">
    <outPort name="eXistOPort" protocol="HTTP" da
taType="TDBStation"/>
  </microservice>
  <microservice name="                  service="P
rocessingOEE" imgBase=                ssembly_st
ation">
    <inPort name="Asse                  protocol
="HTTP" dataType="TDBS                  00"/>
    <outPort name="Ass                 ' protocol
="HTTP" dataType="TSta
  </microservice>
  <microservi                                    O
perator" imgB                                    t
omization="{'
    <inPort name="OEEEventsIPort" protocol="HTTP"
dataType="TStationOEE" number="7000"/>
    <outPort name="OEEEventsOPort" protocol="HTT
P" dataType="TDBOEE"/>
  </microservice>
  <microservice name="Influx" service="StoreOEEDa
ta" imgBase="ekhurtado/gcis:influx">
    <inPort name="InfluxIPort" protocol="HTTP" da
taType="TDBOEE" number="8000"/>
  </microservice>
  <channel from="eXistOPort" to="AssemblyStationI
Port"/>
  <channel from="AssemblyStationOPort" to="OEEEve
ntsIPort"/>
  <channel from="OEEEventsOPort" to="InfluxIPor
t"/>
```

Fig. 11. Web view on Node-RED for the design of Fog Applications.

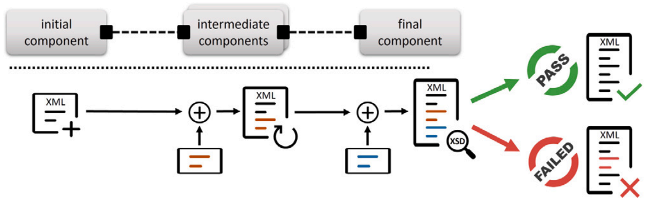


Fig. 12. Automatic generation process of the Application Model.

- The *Data Acquisition Fog application* (orange blocks in Fig. 13) obtains data from the assembly station (1) and stores it for further processing (2).
- The *Data Processing Fog application* (green blocks in Fig. 13) reads the stored data (3), processes it to obtain information about the evolution of the operations performed by the station (4), and finally, saves the results in a database (DB) for later visualization (6). In particular, it calculates the Overall Equipment Effectiveness (OEE), a key performance indicator (KPI) based on the availability, performance and production quality of a machine. This application also generates a warning to plant personnel in case the OEE falls below a predefined threshold, or if it maintains a negative trend (5).

At the Fog level, there are a set of infrastructure resources that provide services that are necessary for the correct operation of the Fog applications (blue blocks in Fig. 13), but do not provide any application logic. These resources are independent of the approach and are not developed within the engineering platform, but should be considered by the Component Programmer when implementing functionalities. In turn, the two Fog applications are connected by a native *XML database* (eXistDB) [64], so that the data acquisition application will store the data of the station which is retrieved by the other application to process it. Finally, there is a *time series database* (influxDB) [65] to store the OEE calculations, which are graphically displayed on a *dashboard* (Grafana dashboard) [66].

The following sections illustrate the steps to follow when utilizing the proposed engineering platform: starting with identifying, designing and developing Fog Components, followed by the design and development of Fog Applications and finalizing with the delivery and operation of the developed Fog Applications.

6.1. Design and development of Fog components

Based on the requirements of the applications, the Component Programmer identifies the necessary components. The components developed are as follows (all their functionalities support the HTTP protocol to send and/or receive data):

- MQTT-HTTP component: it offers functionalities to obtain the data that the assembly station publishes on the MQTT broker.
- eXist component: it provides functionalities to read from and write to the native XML DB.
- AssemblyStation component: it provides functionalities to perform KPI-related analyses.
- OEEevents component: its functionalities give feedback to the plant personnel on the KPI.
- Influx component: its functionalities are related to reading and writing in the time series DB.

In this case study, four component categories are distinguished: source (components that, in general, behave as a data source); events (components that allow reacting to relevant situations); processing (components that perform processing tasks); and sink (components that, in general, are the final destination of data).

Then, the Component Programmer develops the source code that implements the functionalities for each component (Step 2.1) using the parametrization template shown in Fig. 7. After encapsulating the base images using the Dockerfile template illustrated in Fig. 8 (Step 2.2), the Component Programmer stores them in the image repository (Step 2.3), (Google Container Registry [67] in this case study). Finally, the Component Programmer makes use of the XSLT stylesheets to generate the component node and integrates it in the Fog Computing Library.

6.2. Design and development of Fog applications

To better understand the design of the Fog Applications of the case study, Fig. 14 details their data flows. Each microservice is represented by a block with its name on the outside and its service inside. The data type associated to the input and output ports is also shown. Note that infrastructure resources are depicted just to facilitate the understanding of the application operation, as, from the application designer's point of view, they are transparent.

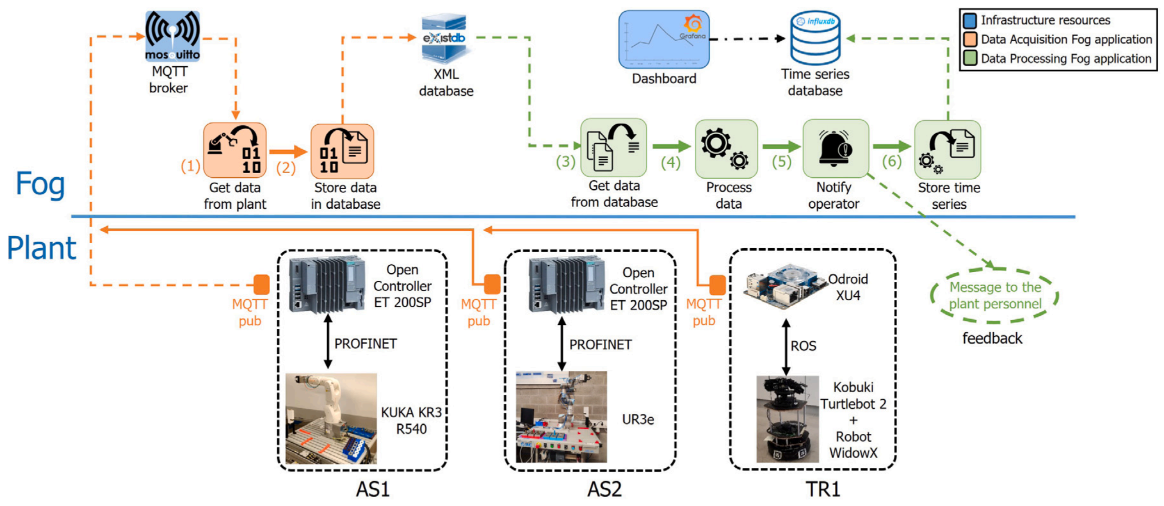


Fig. 13. Functional overview of the case study in the FMS domain.

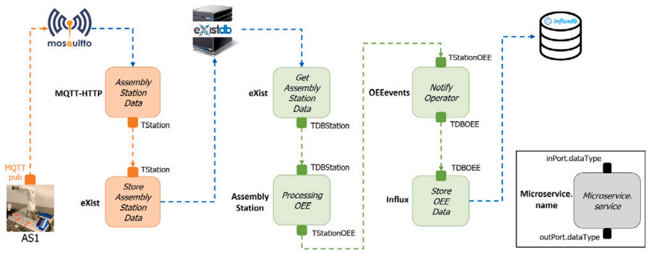


Fig. 14. Workflow of the designed Fog applications for data acquisition and data processing.

The operation of the two applications is as follows. For the Data Acquisition application, every time the AS1 assembly station finishes some operation, it publishes its data in a topic of the MQTT broker. As the MQTT-HTTP microservice is subscribed to this topic, it receives that data and sends it to the eXist microservice, whose *StoreAssemblyStationData* service stores it in the native XML DB. For the Data Processing Fog application, eXist is its first microservice, which, as the *GetAssemblyStationData* service has been selected, obtains the data from the DB, and sends it to the AssemblyStation microservice to process the data received and calculate the OEE. It sends the result to the OEEevents microservice to analyze the result obtained with the *NotifyOperator* service. In case the OEE has fallen below a threshold, it sends a warning to the plant personnel, so that they can take corrective or optimization actions. Eventually, the OEE data is received by the Influx microservice and stored in the time series DB using the *StoreOEEData* service.

As mentioned in Section 5, the Application Designer will use the modified Node-RED toolkit to design and develop these applications. The designer will check if the necessary components are in the Fog Computing Library, and if all of them are available, they will follow the process explained above. Once the workflow is defined, the automatic generation of the Application Model can be requested.

Taking as an example the eXist component, Fig. 15 illustrates the differences on the application development for the same component instantiated in two different applications. In both cases, the same *functional.js* script file is used to obtain different application models but with the same base image (exist_base_image) and different environment variables related to component instantiation: the selected function, the type of the received or sent data and the customization variables, in this case, the ID of the asset. For the Data Acquisition Fog application, the component needs an input port where data is received and, in the case of the Data Processing Fog Application, the component is the first one of the workflow, so it needs an output port to send data.

6.3. Delivery and operation of Fog applications

The proposal offers freedom in the selection of the execution platform of the applications, knowing that for each case the platform's requirements will differ. In this case study Kubernetes has been selected as the container orchestration platform to deliver and operate Fog applications. As mentioned in Section 3.2, Kubernetes is one of the best options for deploying and managing containers.

First, Kubernetes offers several options to deploy containers, but all files use YAML (Yaml Ain't Markup Language) format. For this reason, it is straightforward for the Platform Administrator to transform the XML Application Model to the Application Delivery Model using model-to-model (M2M) transformations, performed through the appModelTransformer.xslt. Kubernetes offers several constructs to deploy and run Docker containers: *Pods*, which are elements where the container is going to create and run (i.e., the microservices), and *Deployments*, to request container deployments in the cluster. In addition, the latter allow the addition of the environment variables to the associated container. Eventually, Kubernetes *Services* enable the communication between microservices.

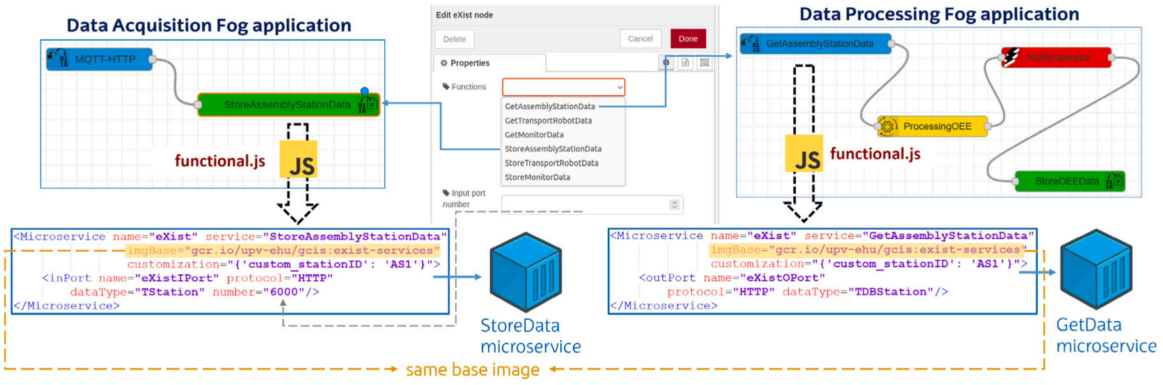
Although Kubernetes itself meets these requirements, it does not have the application concept built in. Moreover, the DevOps approach tries to simplify and automate application deployment. Therefore, based on knowledge of the authors from previous work [48], we propose a Vanilla Kubernetes extension, adding features to achieve these goals. Kubernetes offers some methods to extend the platform, i.e. create *Custom Resources* (CRs) and govern them through Custom Controllers. Kubernetes also allows the integration of meta-models as *Custom Resource Definitions* (CRDs). When the CR is posted in Kubernetes, it is checked against the CRD (the meta-model) to verify its validity.

The following subsections explain how the platform has been modified to automate the entire delivery and operation and how the two Fog Applications of the case study are successfully deployed.

6.3.1. Kubernetes extension to enable automated application operation

As mentioned above, Kubernetes offers tools to integrate meta-models in the platform, using the *Custom Resource Definition*. To follow the meta-models presented in this work, two new resources have been defined as Kubernetes CRs: Application and Microservice.

Each CR has a custom controller associated to it, responsible of managing its lifecycle in Kubernetes. The Application Controller is in charge of processing the Application Delivery Model as follows: a watcher detects when a new *application* resource is posted in the system, which then starts the automatic deployment process. For each



```
<Microservice name="eXist" service="StoreAssemblyStationData"
                imgBase="gcr.io/upv-ehu/gcis:exist-services"
                customization="{'custom_stationID': 'AS1'}">
    <inPort name="eXistIPort" protocol="HTTP"
        dataType="TStation" number="6000"/>
</Microservice>
```

```
<Microservice name="eXist" service="GetAssemblyStationData"
                imgBase="gcr.io/upv-ehu/gcis:exist-services"
                customization="{'custom_stationID': 'AS1'}">
    <outPort name="eXistOPort"
        protocol="HTTP" dataType="TDBStation"/>
</Microservice>
```

Fig. 15. Differences on the application development of a component instantiated in different applications.

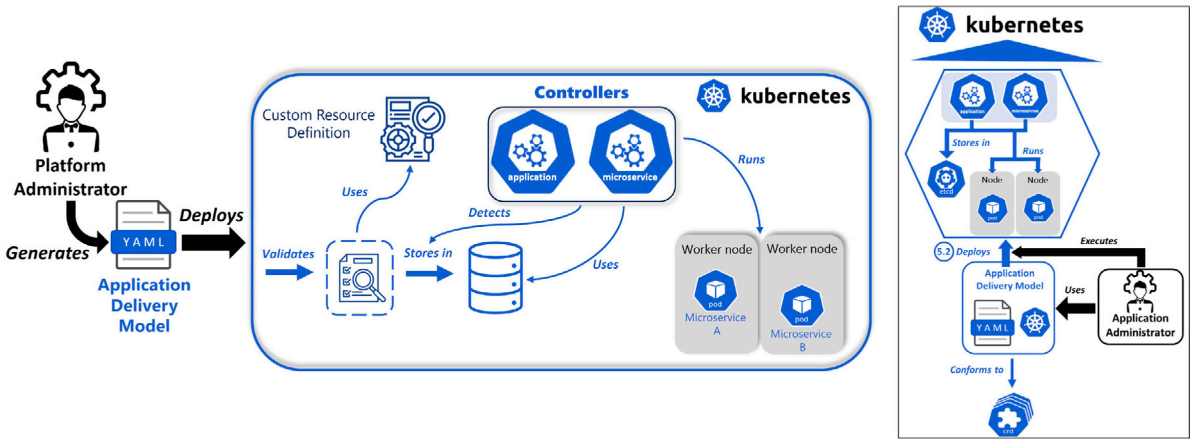


Fig. 16. Automatic Application Delivery Model deployment process using the proposed Kubernetes extension.

microservice, the application controller gets all its information and creates a *microservice* resource. If the microservice needs to be available to others (if it offers a service, has an input port), the controller creates the related *Service* resource, enabling the communication of said microservice.

At the same time, the Microservice Controller is responsible for creating the Deployment. For that, it has a watcher that detects when new microservice resources are created. It creates the Deployment with all the information of the microservice, adding the required environment variables listed in Section 4.2.1 and requests the deployment in the platform. Herein, Kubernetes takes control of the delivery of the application and deploys the microservice inside a Pod, as stated in the Deployment. The application controller will update the status of the applications as their microservices are successfully deployed. Thus, the proposed Kubernetes extension covers the operational part of the DevOps pipeline, automating all the deployment process, illustrated in Fig. 16. As it can be seen, the Application Delivery Model is a single YAML file (an Application CR), greatly simplifying the delivery and operation process.

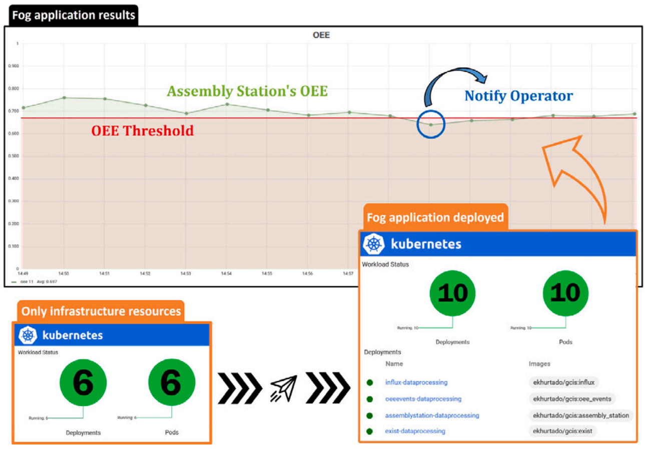
6.3.2. *Kubernetes enabled delivery and operation of Fog applications*

The YAML deployment files obtained are passed to Kubernetes through declarative means using *kubectl* by the Platform Administrator. Kubernetes then assigns each Pod to a node within the cluster and pulls the Docker images from the Google Container Registry. Specifically, K3s, a lightweight and easy to install Kubernetes distribution, has been selected to build the Kubernetes cluster for the case study. From an infrastructure point of view, the cluster is made up of five nodes: a master node, dedicated only to running the K3s, that manages the cluster, and four processing nodes (from now on, workers).

Kubernetes automatically assigns a random string to the Pod name to maintain unique identifiers for the deployed Pods, since more than one pod can be deployed for each deployment (i.e., in case of failure there will be a failed pod and another one with running state). It should be noticed that, while Infrastructure Resources are unique in the system and thus, their identifier is simple, the Deployment name is complex and follows the rule *Application.name-MicroService.name* (the network alias of the service). Each Pod has a unique IP address reachable within the cluster and masked under Kubernetes Services.

Once the Pods are deployed, the container that runs inside them executes the CMD command established in the customized container image. This command executes the code, reading the environment variables, established throughout the whole procedure, and performing the functionality selected.

Fig. 17 shows the deployment of the processing Fog Application presented in this case study. The bottom half reflects the state of the orchestration platform before and after the application deployment. First, only the infrastructure resources (eXistDB, influxDB, MQTT broker and Grafana) and the proposed extension controllers are running (application and microservice controllers). Once the processing application is deployed, it can be seen how the platform state is updated, due to the creation and execution of the 4 new microservices related to that application (eXist-getAssemblyStationData, assemblyStation-processingOEE, OEEevents-notifyOperator, and Influx-Store- OEEData), as presented in Fig. 13 or Fig. 14. When they start executing their functionalities, as the application has been previously validated, it is able to achieve its



Deployments

	Name	Images
●	influx-dataprocessing	ekhurtado/gcis:influx
●	oeeevents-dataprocessing	ekhurtado/gcis:oee_events
●	assemblystation-dataprocessing	ekhurtado/gcis:assembly_station
●	exist-dataprocessing	ekhurtado/gcis:exist

Fig. 17. Status of the Kubernetes cluster before and after the deployment of the processing Fog Application (screenshots from Kubernetes dashboard) and OEE evolution over time (screenshots from Grafana dashboard).

objective, which in this case is the calculation and supervision of the OEE related to the AS1 assembly station. Thus, the OEE calculated from data coming from AS1 is shown at the top of Fig. 17, presented as a snapshot of the Grafana dashboard. To probe the correct operation of the designed applications, a delay was forced during the execution of the different station operations by modifying the robot code to decrease the production rate and hence, the OEE. As it can be observed, when the indicator decreases below a certain given threshold, the NotifyOperator service generates an alert for the operator working in the factory to act accordingly. In this simulated environment, the production rate is reestablished and the OEE goes above the limit threshold established.

7. Conclusions and future work

The smart manufacturing domain is characterized by domain experts that act as application designers who are not experts in application component development or application deployment. In recent times, this activity is being joined by the need to define analytics applications that process plant data in real-time, a task that is usually performed in the Fog. These analytics applications are often tied to information technology systems (IT systems) rather than operational technology systems (OT systems), which results in additional challenges for application designers in the Smart Manufacturing domain. For instance, concepts (such as microservices and containers) that do not exist in plant development standards (such as IEC 61131-3) are introduced. In addition, application designers must deal with the problems that these new concepts already bring with them in the development and runtime phases. In this sense, the reuse of services promoted by the microservices architectural paradigm has led to a great proliferation in the number of microservices, which makes it difficult to discover them and identify their functionalities during development. Moreover, composing complex applications based on microservices also creates reliability, performance, and security challenges during runtime, which should be taken into account in their deployment (e.g., with instructions for the orchestrator to scale the application if necessary).

Although Cloud Engineering solutions are still evolving, in the Smart Manufacturing domain there are no approaches, to the author's knowledge, that allow the necessary separation of concerns to abstract the technical aspects of application component development and delivery from those related to Fog application design, as well as to abstract the latter from the deployment of Fog applications. For these reasons, this work proposes a platform for the composition of industrial Fog applications, aligned with the precepts of OpenFog, that automates containerized application component delivery in a component library, and application deployment using a container orchestrator. This platform provides a solid foundation that allows it to be used without the need to have a deep knowledge of the underlying technologies related to Fog applications. In the opinion of the authors, the novelty of this proposal resides in two aspects: on the one hand, in providing a methodological support that embeds MDE techniques, which innately provide the mechanisms to enable the cooperation between the platform users; on the other hand, in providing a technological support that adopts the Platform Engineering approach, which aims to provision a toolkit that embeds the methodology and all the required functionalities, acting as a common interface to the DevOps infrastructure.

The platform requires an initial setup that involves time and effort to implement but provides a solid separation of concerns between the stakeholders. Work overload and restrictions of the proposal mainly affect the Component Programmers, who are forced to parameterize the source code of the Fog Components. However, this constraint also provides advantages as Component Programmers are free to arrange source code as they need. The possibility of developing several functionalities within the same code could be useful in the case of the same functionality that can communicate through different protocols. These functionalities can be programmed in the same Fog Component, making it possible to reuse functional parts of the source code. Furthermore, Component Programmers must create the corresponding component models and generate the Fog Computing Library nodes. This improves the independence among different groups of Component Programmers who can collaborate through the component models, without having to access the source code. Additionally, the Component Programmer is provided with the stylesheets needed to execute the M2T transformations necessary to obtain valid library nodes. As high-performance and real-time requirements become more relevant, as part of the future work, the Component development process can be optimized and restricted to allow for real-time container deployment.

The platform uses Docker to create containers and Kubernetes as the container orchestrator that manages the deployment of containers and controls their lifecycle, delegating most of the Ops side of the DevOps pipeline on the orchestrator. It should be noted that the proposed methodology is agnostic of the container orchestrator of choice and that in case of modifying the orchestrator, only the stylesheets should be modified. In addition to the advantages offered by Kubernetes itself, the extension of this platform adds benefits such as automating the entire process of deploying Fog Applications as it can generate the microservices and enable the communication between them from the application delivery model. Furthermore, the extension allows the integration of the proposed application meta-model into the platform. In the near future, the platform could be extended to include scalability concerns within the orchestrator itself, augmenting the reusability and adaptability of the designed Fog Applications.

As the proposal is based on MDE techniques, the validity of the Fog Application deployed in Kubernetes is ensured. Once the Fog Components and the Fog Computing Library are developed, the platform eases the design and automates the development of Fog Applications. Therefore, the design and development of Fog Applications requires little effort for the Application Designer. Hence, the separation of concerns between Component Programmers and Application Designers is achieved, while at the same time favoring the reuse of the Fog Components when developing the applications. In addition, the use of the graphical environment abstracts the Application Designer from the design and implementation details of the needed Fog Components, achieving a true separation of concerns, which is essential to reduce the complexity at design level. Furthermore, Application Designers take advantage of the graphical potential and customization capacity of Node-RED. As a result, Application Designers are provided with a graphical interface for applications design from which application development is automated. Finally, although the platform was originally conceived for guiding and providing support in the design and development of microservice-based Fog Applications, the use of MDE techniques allows it to be adopted in similar domains by customizing the meta-models and the stylesheets.

A limitation of the proposal is that, in the case of instantiating the same Fog Component in different applications (e.g., the eXist Fog Component), as many containers as instantiated Fog Components are created. This leads to a waste of resources since the same container could be shared by all applications. Therefore, using the potential of the extension of Kubernetes, the next steps will focus on the development of a Fog architecture that is able to manage the lifecycle of such applications, making it possible for several applications to use the services offered by a containerized microservice. It will also consider the reactivity of Fog applications to feed back into the manufacturing process, beyond simply alerting plant workers.

Funding

This work was financed by the project RTI2018-096116-B-I00 funded by MCIN/AEI/10.13039/501100011033/ and funded by FEDER Una manera de hacer Europa, by the project PES18/48 funded by UPV/EHU, by Open Access funding provided by University of Basque Country, Spain and by the PhD fellowship granted under the frame of the PIF 2022 call funded by the University of the Basque Country (UPV/EHU), Spain, grant number PIF22/188.

CRediT authorship contribution statement

Julen Cuadra: Writing – original draft, Visualization, Validation, Investigation, Conceptualization. **Ekaitz Hurtado:** Writing – original draft, Visualization, Validation, Software, Methodology, Investigation. **Isabel Sarachaga:** Writing – review & editing, Supervision, Investigation. **Elisabet Estévez:** Writing – review & editing, Methodology. **Oskar Casquero:** Writing – review & editing, Supervision, Resources, Project administration, Funding acquisition, Conceptualization. **Aintzane Armentia:** Writing – review & editing, Supervision, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared my code at the manuscript as a GitHub repository.

Appendix A. List of abbreviations

See Table A.1.

Appendix B. Supplementary data

The implementation code of the proposed platform is available at https://github.com/ekhurtado/GCIS_MDE_engineering_platform. The authors also make the required files and stylesheets available on the GitHub repository. Besides, a graphical support is developed to show the use and operation of the proposed platform. This support is available as a video in the following link: https://www.youtube.com/playlist?list=PLs6bFF_iqW3G8AiVVMPi-SpupCvgmYsyB.

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.future.2024.03.053.

Table A.1
List of abbreviations.

Abbreviation	Definition
API	Application Programming Interface
CBSE	Component Based Software Engineering
CR	Custom Resource
CRD	Custom Resource Definition
CSAR	Cloud Service ARchives
DDF	Distributed DataFlow
DSL	Domain Specific Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
KPI	Key Performance Indicator
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MMR	Mobile Manipulation Robots
MQTT	Message Queuing Telemetry Transport
M2M	Model-to-Model
M2T	Model-to-Text
OAM	Open Application Model
OEE	Overall Equipment Effectiveness
PLC	Programmable Logic Controller
TOSCA	Topology and Orchestration Specification for Cloud Applications
UML	Unified Modeling Language
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformations
YAML	Yaml Ain't Markup Language

References

[1] P. Bellavista, A. Zanni, Feasibility of fog computing deployment based on Docker containerization over RaspberryPi, in: 18th International Conference on Distributed Computing and Networking, ACM, Hyderabad, India, 2017, pp. 1–10, http://dx.doi.org/10.1145/3007748.3007777.

[2] J. Harjuhahto, V. Hirvisalo, Positioning fog computing for smart manufacturing, 2022, http://dx.doi.org/10.48550/arXiv.2205.10860, arXiv:2205.10860 [cs].

[3] D. Bouhalouan, B. Nachet, A. Adla, Knowledge-Intensive decision support system for manufacturing equipment maintenance, J. Digit. Inf. Manage. 18 (3) (2020) 85, http://dx.doi.org/10.6025/jdim/2020/18/3/85-98.

[4] E. Hurtado, A. López, A. Armentia, I. Sarachaga, O. Casquero, E. Estevez, M. Marcos, On the development of fog-edge feedback applications, in: 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE), 2021, p. 2.

[5] A. Seitz, D. Henze, D. Miehle, B. Bruegge, J. Nickles, M. Sauer, Fog computing as enabler for blockchain-based IIoT app marketplaces - A case study, in: 2018 Fifth International Conference on Internet of Things: Systems, Management and Security, IEEE, Valencia, Spain, 2018, pp. 182–188, http://dx.doi.org/10.1109/IoTSMS.2018.8554484.

[6] S. Wang, J. Wan, D. Li, C. Zhang, Implementing smart factory of industrie 4.0: An outlook, Int. J. Distrib. Sens. Netw. 12 (1) (2016) 1–10, http://dx.doi.org/10.1155/2016/3159805.

[7] S. Kitanov, T. Janevski, Fog computing orchestration based on network latency, J. Multimedia Process. Technol. 12 (4) (2021) http://dx.doi.org/10.6025/jmpt/2021/12/4/125-131.

[8] J. Alonso, L. Orue-Echevarria, M. Huarte, CloudOps: Towards the operationalization of the cloud continuum: Concepts, challenges and a reference framework, Appl. Sci. 12 (9) (2022) 4347, http://dx.doi.org/10.3390/app12094347, Number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

[9] O. Akrivopoulos, I. Chatzigiannakis, C. Tselios, A. Antoniou, On the deployment of healthcare applications over fog computing infrastructure, in: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), IEEE, Turin, Italy, 2017, pp. 288–293, http://dx.doi.org/10.1109/COMPSAC.2017.178.

[10] A. Barron, D.D. Sanchez-Gallegos, D. Carrizales-Espinoza, J.L. Gonzalez-Compean, M. Morales-Sandoval, On the efficient delivery and storage of IoT data in edge–fog–cloud environments, Sensors 22 (18) (2022) 7016, http://dx.doi.org/10.3390/s22187016, Number: 18 Publisher: Multidisciplinary Digital Publishing Institute.

[11] I. Stojmenovic, S. Wen, The fog computing paradigm: Scenarios and security issues, in: 2014 Federated Conference on Computer Science and Information Systems, 2014, pp. 1–8, http://dx.doi.org/10.15439/2014F503.

[12] J. Singh, P. Singh, S.S. Gill, Fog computing: A taxonomy, systematic review, current trends and research challenges, J. Parallel Distrib. Comput. 157 (2021) 56–85, http://dx.doi.org/10.1016/j.jpdc.2021.06.005.

[13] C.A. Lee, Cloud federation management and beyond: Requirements, relevant standards, and gaps, IEEE Cloud Comput. 3 (1) (2016) 42–49, http://dx.doi.org/10.1109/MCC.2016.15.

[14] IEEE Standard Association, IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing, IEEE Std 1934-2018, 2018, pp. 1–176, http://dx.doi.org/10.1109/IEEESTD.2018.8423800.

[15] G. Mangiaracina, P. Plebani, M. Salnitri, M. Vitali, Efficient data as a service in fog computing: An adaptive multi-agent based approach, IEEE Trans. Cloud Comput. (2022) 1–18, http://dx.doi.org/10.1109/TCC.2022.3220811.

[16] J. Kosińska, G. Brotoń, M. Tobiasz, Knowledge representation of the state of a cloud-native application, Int. J. Softw. Tools Technol. Transf. (2023) http://dx.doi.org/10.1007/s10009-023-00705-2.

[17] M. Waseem, P. Liang, M. Shahin, A. Di Salle, G. Márquez, Design, monitoring, and testing of microservices systems: The practitioners' perspective, J. Syst. Softw. 182 (2021) 111061, http://dx.doi.org/10.1016/j.jss.2021.111061.

[18] O. Zimmermann, Microservices tenets: Agile approach to service development and deployment, Comput. Sci. - Res. Dev. 32 (2016) http://dx.doi.org/10.1007/s00450-016-0337-0.

[19] J. Dobaj, J. Iber, M. Krisper, C. Kreiner, A microservice architecture for the industrial Internet-of-Things, in: Proceedings of the 23rd European Conference on Pattern Languages of Programs, ACM, Irsee Germany, 2018, pp. 1–15, http://dx.doi.org/10.1145/3282308.3282320.

[20] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, M. Villari, Open issues in scheduling microservices in the cloud, IEEE Cloud Comput. 3 (5) (2016) 81–88, http://dx.doi.org/10.1109/MCC.2016.112, Conference Name: IEEE Cloud Computing.

[21] P. Pop, B. Zarrin, M. Barzegaran, S. Schulte, S. Punnekkat, J. Ruh, W. Steiner, The FORA fog computing platform for industrial IoT, Inf. Syst. 98 (2021) 101727, http://dx.doi.org/10.1016/j.is.2021.101727.

[22] R. Buyya, S. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. Vaquero, M. Netto, A. Toosi, M. Rodriguez, I. Llorente, S. Di Vimercati, P. Samarati, D. Milojicic, C. Varela, R. Bahsoon, M. De Assuncao, O. Rana, W. Zhou, H. Jin, W. Gentzsch, A. Zomaya, H. Shen, A manifesto for future generation cloud computing: Research directions for the next decade, ACM Comput. Surv. 51 (5) (2019) http://dx.doi.org/10.1145/3241737.

[23] OASIS, TOSCA version 2.0, 2023, https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html.

[24] OpenFog Consortium, OpenFog reference architecturee, 2017, https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf.

[25] N. Giang, M. Blackstock, R. Lea, Developing IoT applications in the fog: a distributed dataflow approach, in: 2015 5th International Conference on the Internet of Things, IOT, 2015, http://dx.doi.org/10.1109/IOT.2015.7356560.

[26] N.K. Giang, R. Lea, V.C. Leung, Developing applications in large scale, dynamic fog computing: A case study, Softw. - Pract. Exp. 50 (5) (2020) 519–532, http://dx.doi.org/10.1002/spe.2695.

[27] S.W. Kum, J. Moon, T.-B. Lim, Design of fog computing based IoT application architecture, in: 2017 IEEE 7th International Conference on Consumer Electronics - Berlin (ICCE-Berlin), IEEE, Berlin, Germany, 2017, pp. 88–89, http://dx.doi.org/10.1109/ICCE-Berlin.2017.8210598.

[28] S. Taherizadeh, V. Stankovski, Incremental learning from multi-level monitoring data and its application to component based software engineering, in: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), IEEE, Turin, 2017, pp. 378–383, http://dx.doi.org/10.1109/COMPSAC.2017.148.

[29] R. Dintén, P.L. Martínez, M. Zorrilla, Arquitectura de referencia para el diseño y desarrollo de aplicaciones para la Industria 4.0, Rev. Iberoam. Autom. Inform. Ind. 18 (3) (2021) 300–311, http://dx.doi.org/10.4995/riai.2021.14532, Number: 3.

[30] N. Ferry, H. Song, A. Rossini, F. Chauvel, A. Solberg, CloudMF: Applying MDE to tame the complexity of managing multi-cloud applications, in: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, IEEE, London, 2014, pp. 269–277, http://dx.doi.org/10.1109/UCC.2014.36.

[31] V. Cortellessa, D. Di Pompeo, R. Eramo, M. Tucci, A model-driven approach for continuous performance engineering in microservice-based systems, J. Syst. Softw. 183 (2022) 111084, http://dx.doi.org/10.1016/j.jss.2021.111084.

[32] W.P. Luz, G. Pinto, R. Bonifácio, Adopting DevOps in the real world: A theory, a model, and a case study, J. Syst. Softw. 157 (2019) 110384, http://dx.doi.org/10.1016/j.jss.2019.07.083.

[33] H. Dursun, Full spec software via platform engineering: Transition from bolting-on to building-in, in: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, ACM, Oulu Finland, 2023, pp. 172–175, http://dx.doi.org/10.1145/3593434.3593440.

[34] I.-C. Donca, O.P. Stan, M. Misaros, D. Gota, L. Miclea, Method for continuous integration and deployment using a pipeline generator for Agile software projects, Sensors 22 (12) (2022) 4637, http://dx.doi.org/10.3390/s22124637.

[35] Docker, Docker, 2024, https://www.docker.com/.

[36] C. McLuckie, J. Beda, B. Burns, Kubernetes, 2024, https://kubernetes.io/, v1.29.

[37] I. Aldalur, A. Arrieta, A. Agirre, G. Sagardui, M. Arratibel, A microservice-based framework for multi-level testing of cyber-physical systems, Softw. Qual. J. (2023) http://dx.doi.org/10.1007/s11219-023-09639-z.

[38] M. Ugarte Querejeta, L. Etxeberria, G. Sagardui, Towards a DevOps approach in cyber physical production systems using digital twins, in: A. Casimiro, F. Ortmeier, E. Schoitsch, F. Bitsch, P. Ferreira (Eds.), Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops, in: Series Title: Lecture Notes in Computer Science, Vol. 12235, Springer International Publishing, Cham, 2020, pp. 205–216, http://dx.doi.org/10.1007/978-3-030-55583-2_15.

[39] J. Kosińska, K. Zieliński, Autonomic management framework for cloud-native applications, J. Grid Comput. 18 (4) (2020) 779–796, http://dx.doi.org/10.1007/s10723-020-09532-0.

[40] OpenJS Foundation & Contributors, Node-RED, 2024, https://nodered.org, v3.1.5.

[41] H. Gupta, A. Vahid Dastjerdi, S.K. Ghosh, R. Buyya, IFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments, Softw. - Pract. Exp. 47 (9) (2017) 1275–1296, http://dx.doi.org/10.1002/spe.2509, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2509.

[42] I. Alfonso, K. Garces, H. Castro, J. Cabot, Modeling self-adaptative IoT architectures, in: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, Fukuoka, Japan, 2021, pp. 761–766, http://dx.doi.org/10.1109/MODELS-C53483.2021.00122.

[43] D. Perez-Palacin, J. Merseguer, J.I. Requeno, M. Guerriero, E. Di Nitto, D.A. Tamburri, A UML profile for the design, quality assessment and deployment of data-intensive applications, Softw. Syst. Model. 18 (6) (2019) 3577–3614, http://dx.doi.org/10.1007/s10270-019-00730-3.

[44] M. Bogo, J. Soldani, D. Neri, A. Brogi, Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes, Softw. - Pract. Exp. 50 (9) (2020) 1793–1821, http://dx.doi.org/10.1002/spe.2848, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2848.

[45] Y. Wang, C. Lee, S. Ren, E. Kim, S. Chung, Enabling role-based orchestration for cloud applications, Appl. Sci. 11 (14) (2021) 6656, http://dx.doi.org/10.3390/app11146656, Number: 14 Publisher: Multidisciplinary Digital Publishing Institute.

[46] Open application model, 2023, https://github.com/oam-dev/spec.

[47] A. Orive, A. Agirre, H.-L. Truong, I. Sarachaga, M. Marcos, Quality of service aware orchestration for cloud–edge continuum applications, Sensors 22 (5) (2022) 1755, http://dx.doi.org/10.3390/s22051755.

[48] J. Cuadra, E. Hurtado, F. Pérez, O. Casquero, A. Armentia, OpenFog-compliant application-aware platform: A kubernetes extension, Appl. Sci. 13 (14) (2023) 8363, http://dx.doi.org/10.3390/app13148363, Number: 14 Publisher: Multidisciplinary Digital Publishing Institute.

[49] N. Petrovic, M. Tosic, SMADA-Fog: Semantic model driven approach to deployment and adaptivity in fog computing, Simul. Model. Pract. Theory 101 (2020) 102033, http://dx.doi.org/10.1016/j.simpat.2019.102033.

[50] I. Kumara, P. Mundt, K. Tokmakov, D. Radolović, A. Maslennikov, R.S. González, J.F. Fabeiro, G. Quattrocchi, K. Meth, E. Di Nitto, D.A. Tamburri, W.-J. Van Den Heuvel, G. Meditskos, SODALITE@RT: Orchestrating applications on Cloud-Edge infrastructures, J. Grid Comput. 19 (3) (2021) 29, http://dx.doi.org/10.1007/s10723-021-09572-0.

[51] C. Avasalcai, B. Zarrin, S. Dustdar, EdgeFlow—Developing and deploying latency-sensitive IoT edge applications, IEEE Internet Things J. 9 (5) (2022) 3877–3888, http://dx.doi.org/10.1109/JIOT.2021.3101449.

[52] H. Song, R. Dautov, N. Ferry, A. Solberg, F. Fleurey, Model-based fleet deployment in the IoT–edge–cloud continuum, Softw. Syst. Model. 21 (5) (2022) 1931–1956, http://dx.doi.org/10.1007/s10270-022-01006-z.

[53] T. Vale, I. Crnkovic, E.S. de Almeida, P.A.d.M. Silveira Neto, Y.C. Cavalcanti, S.R.d. Meira, Twenty-eight years of component-based software engineering, J. Syst. Softw. 111 (2016) 128–148, http://dx.doi.org/10.1016/j.jss.2015.09.019.

[54] Cloud native landscape, 2024, https://landscape.cncf.io/.

[55] W3C, W3C XML schema definition language (XSD) 1.1 part 1: Structures, 2012.

[56] W3C, XSL transformations (XSLT) version 2.0 (second edition), 2021, https://www.w3.org/TR/2021/REC-xslt20-20210330/.

[57] L.A. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, A kubernetes controller for managing the availability of elastic microservice based stateful applications, J. Syst. Softw. 175 (2021) 110924, http://dx.doi.org/10.1016/j.jss.2021.110924.

[58] R. Fayos-Jordan, S. Felici-Castell, J. Segura-Garcia, A. Pastor-Aparicio, J. Lopez-Ballester, Elastic computing in the Fog on Internet of Things to improve the performance of low cost nodes, Electronics 8 (12) (2019) http://dx.doi.org/10.3390/electronics8121489.

[59] R. Fayos-Jordan, S. Felici-Castell, J. Segura-Garcia, J. Lopez-Ballester, M. Cobos, Performance comparison of container orchestration platforms with low cost devices in the fog, assisting Internet of Things applications, J. Netw. Comput. Appl. 169 (2020) http://dx.doi.org/10.1016/j.jnca.2020.102788.

[60] H. Kang, M. Le, S. Tao, Container and microservice driven design for cloud infrastructure DevOps, in: 2016 IEEE International Conference on Cloud Engineering (IC2E), 2016, pp. 202–211, http://dx.doi.org/10.1109/IC2E.2016.26.

[61] F. Xhafa, B. Kilic, P. Krause, Evaluation of IoT stream processing at edge computing layer for semantic data enrichment, Future Gener. Comput. Syst. 105 (2020) 730–736, http://dx.doi.org/10.1016/j.future.2019.12.031.

[62] B. Costa, J. Bachiega, L.R. de Carvalho, A.P.F. Araujo, Orchestration in fog computing: A comprehensive survey, ACM Comput. Surv. 55 (2) (2022) 29:1–29:34, http://dx.doi.org/10.1145/3486221.

[63] OASIS, MQTT - The standard for IoT messaging, 2019, https://mqtt.org/.

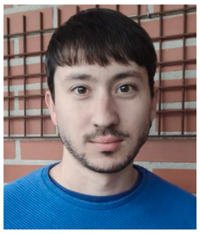
[64] W. Meier, eXist-db - The open source native XML database, 2023, http://exist-db.org/exist/apps/homepage/index.html, v6.2.0.

[65] P. Dix, Influxdb, 2024, https://www.influxdata.com/, v2.7.5.

[66] T. Ödegaard, Grafana dashboards, 2024, https://grafana.com/grafana/dashboards/, v10.4.

[67] Google container registry, 2024, https://cloud.google.com/container-registry?hl=es.

Julen Cuadra graduated in Industrial Technology Engineering and obtained a master's degree in Industrial Engineering at the Faculty of Engineering in Bilbao (UPV/EHU) in 2019 and 2021, respectively. In September 2021, he started a Ph.D. program in control, automation, and robotics engineering at the Faculty of Engineering in Bilbao. As a researcher, he is part of the GCIS (Grupo de Control e Integración de Sistemas) research group at the Department of Systems Engineering and Automatic Control, where he is focused on the development and integration of Fog Computing solutions.


Ekaitz Hurtado graduated in IT from the Faculty of Engineering in 2020. In September 2021, he started the Control, Automation and Robotics Engineering master's degree where he developed the Final Project focused on the development and integration of the Fog Computing paradigms in industrial solutions. In September 2023, he started a Ph.D. program in control, automation, and robotics engineering. He has been part of the GCIS (Grupo de Control e Integración de Sistemas) research group at the Department of Systems Engineering and Automatic Control from September 2021 to February 2022 as contracted researcher and from September 2023 as a researcher.

Dr. Isabel Sarachaga is Permanent Associate Professor in Automatic Control and Systems Engineering at the University of the Basque Country (UPV/EHU). She graduated with B.Sc. in Computer Science from the University of Deusto in 1990 and obtained her Ph.D. degree from the same University in 1999. Since 2001, she has been a member of the Systems Control and Integration research group (GCIS) of the UPV/EHU. Currently, her main research interests deal with the application of the Model Driven Engineering paradigm to industrial control systems and reconfigurable distributed applications in the context of Industry 4.0.

Dr. Elisabet Estévez is a lecturer in the Electronics and Automation Engineering department at the University of Jaen. She graduated in Telecommunications Engineering from the University of the Basque Country (UPV/EHU) in 2002. She obtained her Ph.D. in Automatic Control (UPV/EHU) in 2007. She is co-author of over 120 technical articles in international journals and conference proceedings in the field of industrial control systems. She participated in research projects funded both nationally and by the European Union RD. Her expertise falls mainly within the field of industrial control. She has also been a reviewer for several conferences and technical journals.

Dr. Oskar Casquero received the B.S. degree in telecommunication engineering and the Ph.D. degree in engineering, in 2003 and 2013, respectively. From 2004 to 2007, he worked as an IT Architecture Analyst at the Virtual Campus of the University of the Basque Country. Since 2007, he has been working as an Assistant Professor with the Systems Engineering and Automatic Control Department, currently at the Faculty of Engineering, Bilbao. He investigates on smart and flexible manufacturing systems using digital twins, model-driven engineering, multi-agent systems and cloud computing technologies. Dr. Casquero collaborates as a reviewer with several indexed journals and international conferences.

Dr. Aintzane Armentia graduated in Telecommunications Engineering by the University of the Basque Country (UPV/EHU) in 2001. After 6 years of industrial experience, she started to work as a researcher in the Department of Automatic Control and Systems Engineering of the same university. In 2011 she obtained her M.Sc. degree in Control, Automation and Robotics Engineering, and in 2016 she obtained her Ph.D. degree. She is currently an assistant professor in the same department. Her research interest is focused on Model Based Engineering, multiagent systems, cloud computing technologies and smart manufacturing systems.