Article

OpenFog-Compliant Application-Aware Platform: A Kubernetes Extension

Julen Cuadra *, Ekaitz Hurtado, Federico Pérez, Oskar Casquero * and Aintzane Armentia

Systems Engineering and Automatic Control Department, University of the Basque Country (UPV/EHU), 48013 Bilbao, Spain; ekaitz.hurtado@ehu.eus (E.H.); federico.perez@ehu.eus (F.P.); aintzane.armentia@ehu.eus (A.A.)

* Correspondence: julen.cuadra@ehu.eus (J.C.); oskar.casquero@ehu.eus (O.C.); Tel.: +34-(94)-6014213 (J.C.); +34-(94)-6014459 (O.C.)

Abstract: Distributed computing paradigms have evolved towards low latency and highly virtualized environments. Fog Computing, as its latest iteration, enables the usage of Cloud-like services closer to the generators and consumers of data. The processing in this layer is performed by Fog Applications, which are decomposed into smaller components following the microservice paradigm and encapsulated into containers. Current state-of-the-art container orchestrators can manage hundreds of simultaneous containers. However, Kubernetes, being the de facto standard, does not consider the application itself as a top-level entity, which limits its orchestration capabilities. This raises the need to rearchitect Kubernetes to benefit from application-awareness, which refers to an orchestration method optimized for managing the applications and the set of components that comprise them. Thus, this paper proposes an application-aware and OpenFog-compliant architecture that manages applications as first-level entities during their lifecycle. Furthermore, the proposed architecture allows the definition of organizational structures to group subordinated applications based on user-defined hierarchies. This logical structuring makes it possible to outline how orchestration should be shaped to reflect the operating model of a system or an organization. The proposed architecture is implemented as a Kubernetes extension and provided as an operator.

Keywords: Fog Computing; OpenFog; Kubernetes; container; microservice; application-aware

Citation: Cuadra, J.; Hurtado, E.; Pérez, F.; Casquero, O.; Armentia, A. OpenFog-Compliant Application-Aware Platform: A Kubernetes Extension. *Appl. Sci.* **2023**, *13*, 8363. https://doi.org/10.3390/app13148363

Academic Editor: Juan Francisco De Paz Santana

Received: 19 June 2023
Revised: 12 July 2023
Accepted: 17 July 2023
Published: 19 July 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Fog Computing has emerged as a distributed computing paradigm that pretends to solve some of the Cloud Computing issues [1–4], as it brings the services typically offered by the Cloud closer to the devices that produce the data, improving data security and networking performance [5,6]. For example, in the Smart Manufacturing domain, Programmable Logic Controllers (PLCs) commonly used in factories are strongly adapted to execute sequential and repeating applications that control and monitor a machine or a process. However, PLCs usually lack the computing, storage and networking capabilities needed to execute analytical applications. In this context, Fog Computing allows for an improvement of the responsiveness and task conformance of the analytic applications and enables feedback to the productive process.

Being a relatively novel research topic, great efforts have been made to standardize Fog Computing. OpenFog is the most renowned standard, being an IEEE standard since 2018 [7]. OpenFog states that the Fog works in coordination with the Cloud to offer high-level services to the lowest tier of IoT devices. The distinction between Fog Computing and Edge Computing is a debated topic [4,5,8,9]. Although some authors consider that the Fog and the Edge should be treated as equals [1,10,11], this work is aligned with OpenFog and, thus, we refer to the set of heterogeneous and distributed resources between the Cloud and the IoT devices as Fog instead of Edge.

OpenFog defines Fog Applications (herein, applications) as being "*composed of a loosely coupled collection of microservices*" [7] (p. 85). The microservice paradigm used to design applications is based on separating the application logic into smaller independent, executable, scalable and upgradeable elements (i.e., microservices) that work together to achieve the goal of the application [12–14]. Therefore, we align with the OpenFog consideration of Fog applications as a collection of microservices and focus on the separation of concerns between the development of the application components and the design of applications.

To execute microservices, Fog Computing takes advantage of the lightweight virtualization method that containers provide. This way, each microservice can be encapsulated inside of a container and deployed in distributed devices, or Fog Nodes as OpenFog calls them. Handling the numerous containers in a computer cluster and provisioning a communication environment around them requires a tool for container deployment, scaling and management. OpenFog recommends automating all of these actions through an entity commonly known as a container orchestrator. There are different options, Kubernetes being the most popular one [15]. It is estimated that around 70% of the published academic works have used Kubernetes as opposed to other orchestration solutions [16]. Additionally, many of the leading IT companies offer managed Kubernetes versions and other popular orchestration solutions are based on Kubernetes, such as Red Hat's OpenShift [17].

The design of applications and the development and deployment of the components that compose them are usually considered separate processes. When designing Cloud Application, there are two main ongoing projects that are worth mentioning and analyzing on their own:

1. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [18] is a standard that describes the usage of topology templates to define workloads that represent Cloud Applications as a collection of services [19]. The topology template is viewed as a graph of components and the relationships between them [20], where lifecycle dependencies can be established for the components, as utilized in [21].
2. The Open Application Model (OAM) [22] focuses on the modeling of a cloud application as a Directed Acyclic Graph (DAG). They define a Cloud Application as follows: "*A cloud native application is a collection of interrelated, but discrete components (services, tasks, workers) that, when coupled with configuration and instantiated in a suitable runtime infrastructure, together accomplish a unified functional purpose*". The OAM mainly consists of two elements: the OAM component that defines workloads related to certain runtime environments and the OAM application where the OAM components are interconnected.

Despite the differences they may exhibit, these projects have the same goal: to treat the set of components of an application as a functional unit. This has led to the search for synergies between them: in [12], the authors define applications based on TOSCA and transform them into OAM-compatible files.

The design of an application as a DAG helps represent its computations and the communications that interrelate them [23]. Therefore, in the context of Fog Computing, a Fog Application is equivalent to a DAG where the vertices represent the microservices and the edges represent the data exchange between them. Figure 1a shows how the conception of Fog Applications as DAGs can be applied in the manufacturing domain, allowing the design of applications that describe different operations related to the data of each asset in the factory.

However, the DAG concept does not establish relations between applications, although some may refer to the same asset. In fact, the DAGs of Figure 1a could be arranged by a robot, as in Figure 1b, or by Assembly Lines (AL) as in Figure 1c. Organizing DAGs in different ways would provide benefits from the point of view of their management, making it possible to reflect the operating model of a system or an organization. An orchestrator could provide support to help establish relationships between DAGs; but, as far as the authors know, this requirement has not been addressed yet. Kubernetes provides a resource called Deployment that allows for defining a list of containers and accordingly

parameterizing them to deploy an application. A single container, or a group of containers sharing resources, is hosted inside what Kubernetes calls a Pod, the smallest deployable computing unit in Kubernetes. Thus, a Pod represents a microservice. The developer is responsible for translating the design of the application as a collection of microservices to a list of containers in a Deployment. Therefore, although Kubernetes can manage all the microservices that comprise an application, it is not aware of their collective representation as an application. In this sense, since Kubernetes cannot handle the application itself as a first-level entity, neither can it handle the organizational structure of a set of interrelated applications and leverage it for application management.

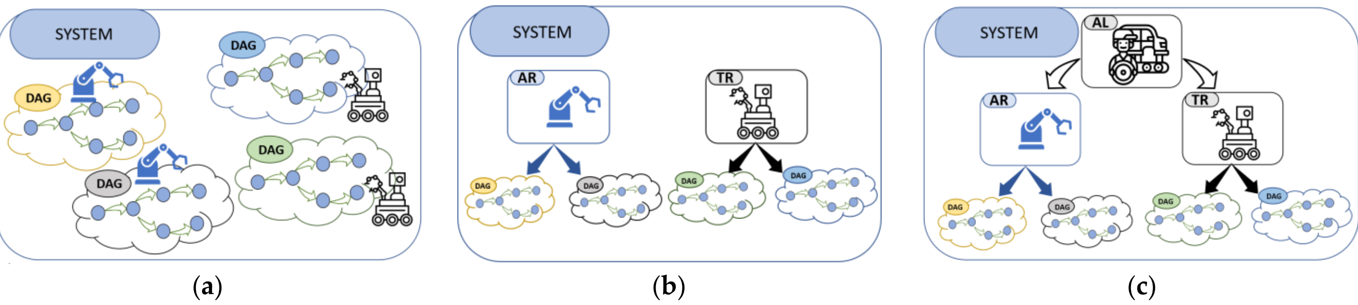

Figure 1. DAG organization according to different criteria: (**a**) Unrelated DAGs in the system; (**b**) DAGs organized by robot, Assembly Robot (AR) and Transport Robot (TR) and (**c**) DAGs organized by Assembly Line (AL). Icons made by Eucalyp and manshagraphics from www.flaticon.com (accessed on 6 June 2023).

In this context, this work focuses on integrating the application concept into container orchestrators, which would enable application-aware platforms; that is, platforms with built-in intelligence that could treat applications as first-level entities, providing automated and more efficient management. Application-aware orchestration may include monitoring the status of the microservices that compose the application as well as managing their lifecycle during the deployment and execution of the application.

This work contributes an application-aware Fog platform aligned with OpenFog. More precisely, the contribution of this paper is twofold: (a) proposal of a novel hierarchical application structure, which extends the typical conception of applications as DAGs, with user-defined N levels from the application management point of view (herein, the so-called "Hierarchical Application Management Structure", HAMS) and (b) integrating the previous proposal in Kubernetes. As a result, Kubernetes will be aware of the user-defined concepts and levels in which the application management hierarchy is structured, managing their execution as if they were native Kubernetes objects. All of this provides an OpenFog Technology Ready platform which, along with the case study presented, classifies as a Technology Readiness Level of four.

The rest of the article is structured as follows. Section 2 presents a survey section that analyzes the contributions as well as limitations of the related state-of-the-art work. Section 3 presents our OpenFog-compliant system architecture and the Kubernetes extension methods for its implementation, followed by a description of the available testbed and the designed applications. Section 4 describes the platform resulting from the implementation of our proposal in Kubernetes. Section 5 discusses the usage of the platform and the deployment of the case study. Finally, Section 6 presents the conclusions and comments on future work.

2. Related Work

As the objective of the paper is twofold, the state of the art is analyzed from the point of view of both contributions: first, the literature regarding Fog Applications is analyzed; then, works related to Kubernetes extensions are described.

2.1. Fog Applications

This section is devoted to the analysis of applications in Fog Computing or in Cloud to Edge Computing solutions. The focus of this analysis is to identify the main way of designing, composing and deploying applications in these distributed environments.

In [12], the authors studied the management of application elasticity in Kubernetes. To do so, they divided applications into small and independent microservices. Each of these microservices represented a single functionality encapsulated in a Pod, managed by Kubernetes. The application itself was represented as a Deployment in Kubernetes and they considered it to make runtime adaptation decisions about the elasticity of the microservices.

Sebrechts [21] focused on modeling the applications and explicitly reflecting the relationship between the services that compose them based on TOSCA concepts. In fact, tools such as Kubernetes do not reflect the connections, and that task requires reverse engineering from the application managers.

The division of applications in microservices was also treated in [20], leaning on TOSCA to reflect the microservices and their communication relationships. Furthermore, the authors focused on the importance of considering the communications between microservices when scheduling individual Pods in Kubernetes. The approach presented in [19] presented a model-driven, role-based application orchestration approach where applications were designed based on TOSCA templates, transformed into OAM-compatible YAML files and deployed in a OAM-extended Kubernetes. This way, the application design could leverage the capabilities of TOSCA modeling and abstract the designer from the implementation in a deployment platform.

In [13], the authors presented an Application-Centric Orchestration Architecture (ACOA) focused on the distributed scheduling of applications in the Cloud-Edge continuum. Following the *Workload Model*, applications are made up of components that comprise executable elements. Components are deployed inside containers and the application topology is represented in channel elements that reflect message transmission between pairs of components, representing applications as directed graphs.

Other works have proposed the division of applications in functions using the serverless pattern [14,24]. Functions are executed in containers and managed by orchestrators.

It is clearly noticeable that microservices are the leading paradigm when designing applications in distributed environments. Furthermore, the design of applications and their deployment are usually considered separate processes and decoupling the design of applications from the development and deployment of the components that compose them is a requirement in the works identified.

2.2. Kubernetes Extensions

The management of applications or their components in distributed environments such as the Fog has been discussed in Section 1. Kubernetes is the leading tool when it comes to container orchestration. As such, many authors inspect the capabilities that Kubernetes offers when considering its extension to accommodate any custom logic that might want to be implemented in the orchestrator.

KubeEdge [25] is an Edge infrastructure built on top of Kubernetes that aims to provide RPC-based communications between microservices deployed at the Edge and the Cloud. A Kubernetes controller is deployed on the Cloud (*EdgeController*) to remotely manage edge nodes, and to allow remote deployment on the Edge from the Cloud. The state of the cluster is synced between the Cloud and the Edge through a sync service and the workloads are managed by a *EdgeCore* agent running on the Edge nodes.

The authors in [12] proposed a hierarchical Kubernetes extension (*Me-kube*) to manage the elasticity of microservice-based applications. Their approach is based on Monitor, Analyze, Plan and Execute (MAPE) control loops: one MAPE loop for each microservice, acting as a Microservice Manager, and one loop for each application, acting as an Application Manager. This allows for the definition of global policies for the management of Applications and different local policies for their microservices. Application Managers determine

the scaling actions to be implemented based on microservice metrics and communicate decisions to the Microservice Managers that execute decisions through the Kubernetes API.

The work in [26] presented a Kubernetes controller (High Availability State Controller) to address availability issues that emerge with the default high availability Kubernetes services to restore Pods. The controller reacts to Pod scaling events and selects an active Pod in charge of providing a certain service, while another Pod is identified as a standby Pod, aware of the active's state and ready to intervene given that the active Pod fails.

In [27], the authors utilized Kubernetes extensions to introduce a new concept (*Dataset*) on Kubernetes, managed by the *Datashim* framework. The Dataset is implemented as a Custom Resource Definition (CRD), and each of these resources acts as a pointer to some data source implemented as a Persistent Volume Claim. They construct a Dataset operator to manage the Dataset resources using the operator-sdk toolkit. This controller then creates a Persistent Volume Claim and a Secret for every Dataset resource.

The authors in [28] presented a management model for the Cloud Native Network Function. The Kubernetes Master Node is extended with a Network Management Controller (a Kubernetes controller) and a Network Management Resource Definition component (a Kubernetes CRD). Kubernetes Worker Nodes, on the other hand, are extended through the *sidecar* concept where two containers are deployed inside a Pod, one of them acts as a proxy (Management Agent) while the other container implements the application logic.

The authors of ACOA [13] implemented their generic architecture over Kubernetes, utilizing the standard Kubernetes extension methods. They implemented an Application Controller and a Component Controller to manage applications and components, enabling an application-centric scheduling approach.

The work in [21] presented the *orcon* orchestrator, built as a Kubernetes extension. They introduced the concepts of relationships, interfaces and roles to the Kubernetes API, implemented as annotations in Kubernetes objects to which orcon services react. To extend the definition of the roles, two CRDs are implemented: ProviderConfig and ConsumerConfig. These resources are managed by the Relations Controller, implementing the Kubernetes Controller pattern.

Other works such as [20,29–33] developed different Kubernetes scheduler extensions, by modifying one of the tasks of the scheduler or even entirely replacing it. Most of these approaches utilize an agent that runs on worker nodes, collects metrics such as latency and feeds these metrics to the scheduler to enable decision making. The approach in [31] goes a step further and creates a Custom Resource NodeProfile to store the resource profile of a given node. This resource is accompanied by a controller (Zeus-manager) that manages the life cycle of the NodeProfile resources and takes rescheduling decisions.

Some efforts have been made to develop orchestration solutions. However, these solutions cannot compete with the coverage of state-of-the-art container orchestrators. Kubernetes extensions offer great capabilities to add custom logic to container orchestration, as in the works reviewed here. Although these works do not necessarily follow our objective of developing an application-aware orchestrator, they utilize Kubernetes to achieve their different orchestration objectives, and, thus, we took them as a reference to guide our solution.

3. Materials and Methods

First, this section presents the conceptualized architecture, as well as the extension of the application concept with the HAMS. Then, the Kubernetes architecture is analyzed, and its extension methods are detailed. Finally, the available testbed is described.

3.1. OpenFog-Compliant Architecture

This section is devoted to presenting the system architecture and its working principles. Figure 2 shows the architecture, based on the three-layer design proposed by OpenFog. The lowest level is composed of IoT devices, that is, sensors and actuators that interact with the physical world. The second layer, or Fog, comprises the devices that receive, process and send data produced by the IoT devices to both the Cloud and back to the IoT layer.

The uppermost layer, or Cloud Computing, is composed of a set of centralized data centers, which provide permanent storage capabilities and store container images in the Container Image Registry (CIR) for their use in the Fog layer. The interaction between the layers is as follows: the IoT device layer utilizes a Fog-IoT Message Hub to send messages to and receive messages in an asynchronous manner from the Fog; the Cloud layer, on the other hand, utilizes a Fog-Cloud Interface to send and receive data from the Fog in a direct and synchronized manner.

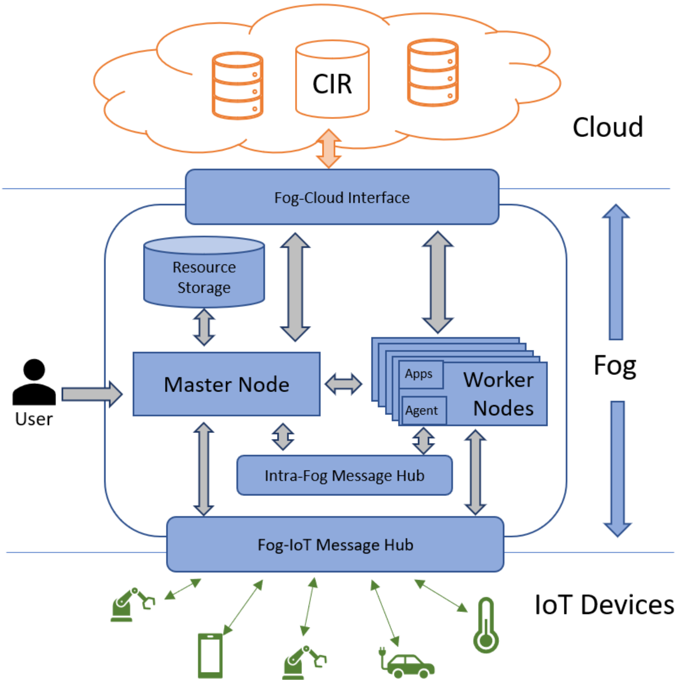

Figure 2. Proposed system architecture.

At the Fog layer, the architecture is based on a Fog Node acting as a Master Node that manages the cluster, and a set of Worker Nodes that perform the necessary processing. The Master Node enables the centralized access to the cluster for external users, manages the dynamic addition of Worker Nodes to the cluster and assigns the workloads to be deployed on each Worker Node. Moreover, the Resource Storage at the Fog layer is centrally accessible through the Master Node. To run containers, both the Master Node and the Worker Nodes need an adequate container runtime engine that enables an environment for container execution.

An agent runs on each Worker Node to locally manage the workloads assigned and provides the necessary network infrastructure for workload communication. The workloads assigned to the Worker Nodes are the microservices that compose an application. In fact, Kubernetes defines workloads as "*an application running on Kubernetes*" [34]. Each microservice runs inside of a container and, to support the decoupling of the microservices, they make use of the Intra-Fog Message Hub to communicate by using a data-oriented protocol. On the other hand, the Master Nodes and the Worker Nodes directly communicate through a request-response protocol, such as HTTP, and can talk to the Cloud and the IoT Devices layers via the Fog-Cloud interface and the Fog-IoT Message Hub, respectively.

The platform resulting from the implementation of this architecture is designed following the principle of Platform as a Service (PaaS). Therefore, the resulting platform, as OpenFog states, "*allows customers to develop, run and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an application*" [7] (p. 14). The platform is composed of applications that represent the workloads deployed on the Fog Nodes. The application logic can be subdivided into application components that work together to accomplish the business objectives of the

application. These components are selected when designing the application and, following OpenFog's philosophy, are considered Application Microservices during runtime. In summary, business logic is designed as functionalities offered by application components and provided as services offered/required by Application Microservices.

To be compliant with OpenFog, the authors propose a software architecture that relies on the Software viewpoint of OpenFog's architecture. This is based on a three-layer model that describes the software that runs on a Fog platform. Figure 3 compares OpenFog's stack to our software architecture stack.

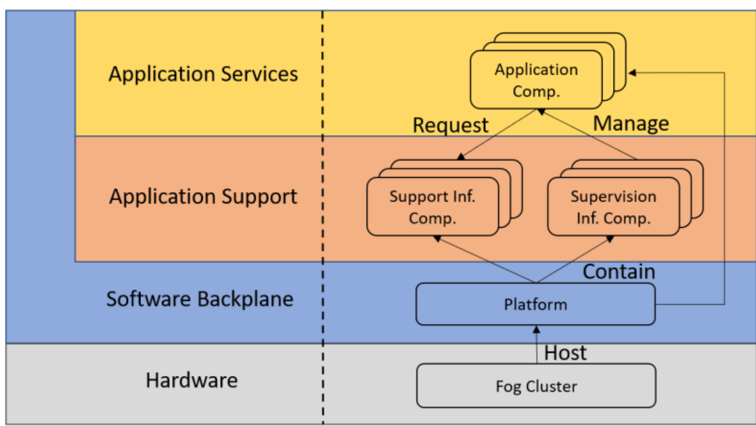

Figure 3. Comparison of OpenFog's Software View (**Left**) and our Software Stack (**Right**).

The bottom layer of Figure 3 brings together the aspects related to the hardware on top of which the Software Backplane (the platform itself) is built. The Application Support services lay on the Software Backplane. These services do not satisfy any business-specific need on their own. However, they provide infrastructure services (i.e., message buses, storage volumes, operation management) to the workloads that do achieve business goals and applications. In our platform, Application Support services are covered by the Infrastructure Components, which are subdivided into Support Infrastructure Components and Supervision Infrastructure Components.

- Support Infrastructure Components offer common services that are deployed once per cluster and assist in operations such as component communication. They are represented during runtime as infrastructure microservices, providing an API for each one of the services they offer.
- Supervision Infrastructure Components offer services for the management of the resources related to the HAMS. These components are responsible for creating the required resources and managing their relationships and their lifecycles.

Therefore, the Software Backplane acts as a control plane (e.g., making global decisions about the cluster, such as starting new Pods or assigning Pods to Nodes), while the Application Support acts as the management plane (e.g., in the manufacturing domain, enabling tasks closer to the applications, such as launching all the DAGs related to a station at boot, or stopping all DAGs related to a line when it stops as production needs decrease).

The uppermost level of the Software viewpoint of OpenFog's architecture is composed of Application Services. These fulfill business goals and requirements and are dependent on the services provided by the lower layers. Therefore, applications make use of Application Services and Application Support services to pursue their goals. In our platform, Application Services are covered by Application Components, represented at runtime by application microservices.

According to their persistence on the platform, application components are separated into Ephemeral and Permanent components. Ephemeral application components have their lifecycle tied to the application to which they belong. They start running at the same time the application deployment is requested and are decommissioned from the platform

at the same time as the application. Permanent components, on the other hand, exist on the system and their deployment can be requested by the users or by the applications themselves. However, their lifecycle is not tied to any application. Thus, permanent application components may exist prior to the deployment of any application and may persist once the application is no longer running. The services provided by these permanent application components may be used at the same time by many applications, resulting in a multitenancy of these components and optimizing the performance of the platform.

Applications deployed on the platform are also categorized according to their lifecycle. Permanent applications pursue the realization of stream processing tasks, where incoming data are constant and some kind of data treatment is required; for example, IoT sensors in continuous processes. Ephemeral applications pursue the realization of batch processing, where data comes in batches or packages, are processed on demand and, upon completion, the application is no longer required. Thus, ephemeral applications are deployed to process some data and decommissioned once the data has been processed. This behavior is inherent to manufacturing processes where production requests come in batches as opposed to a constant manner.

Lastly, the deployment of every concept previously explained in this section is performed leveraging the advantages the container virtualization offers. In our platform, containers encapsulate both application microservices and infrastructure microservices so that these containers can be deployed on the diverse Fog Nodes that form the platform. These nodes are hierarchically structured and their management, as well as the management of the container execution, is delegated to the container orchestrator.

3.2. Hierarchical Application Management Structure

This section presents the Hierarchical Application Management Structure on which our proposal is based: "a hierarchical application structure that extends a DAG by adding N levels defined by application designers from an application management point of view. As previously stated, integrating such an application structure in orchestrators allows for the management of the microservices, the DAGs themselves and the HAMS levels.

The starting point is the two levels usually considered in DAGs (Application and Component), to which N additional levels are added that allow for the management of applications at different levels. Therefore, we define the Application Structure concept as the composition of the two DAG levels and the N HAMS upper levels. In general, each level acts as a manager of the elements of its lower level. As an instance, Figure 4 shows an example of a HAMS where N = 1. An application formed by two components is assigned to a Level 1 Resource.

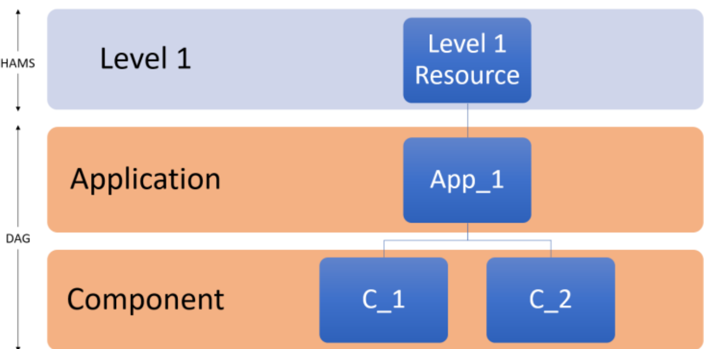

Figure 4. HAMS exemplification where N = 1.

The previous example can be both vertically and horizontally scaled, which increases the management possibilities. Vertical scaling implies adding additional levels to the hierarchy. For example, the previously described Level 1 Resources could be organized in Level 2 Resources, which implies adding a new upper level to the HAMS of Figure 4. Therefore, in this case, the number of HAMS levels (N) is raised to two (see Figure 5).

Horizontal scaling means adding new DAGs or even adding new elements at any of the N levels of the HAMS. Scaling is not limited to adding levels or elements to the levels, but they can also be scaled down; i.e., removed from the system without affecting the global representation of it. Note that the proposed HAMS is flexible enough to allow elements of any of the N levels to exist on the system without being part of any element of its upper level (see rightmost Level 1 Resource in Figure 5).

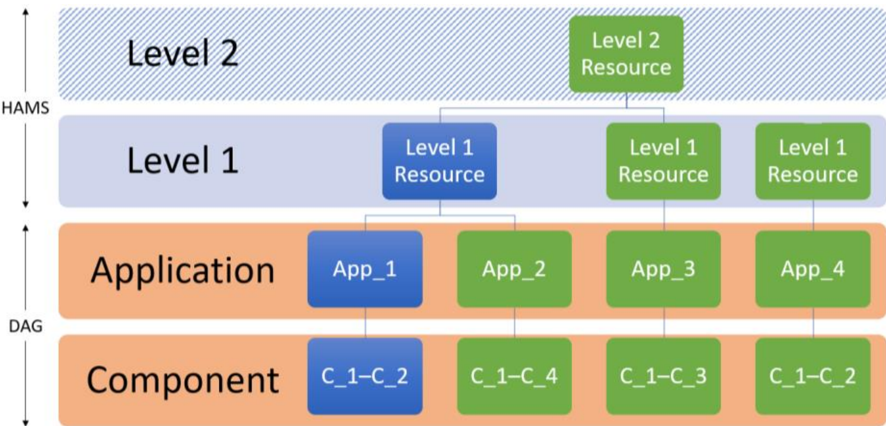

Figure 5. HAMS vertically scaled to where N = 2 and horizontal scalability (highlighted in green) illustrated adding additional Resources and additional Applications.

At runtime, each level of the HAMS is implemented by a Supervision Infrastructure Component aware of the resources of its level and is responsible for the management of the elements of its lower level. Details of the HAMS implementation within the Kubernetes orchestrator can be found in Section 4.2.

3.3. Kubernetes and Its Extension Methods

This section presents the implementation of the previously described architecture. As Kubernetes is the de facto standard when it comes to container orchestration, this work proposes integrating the architecture through standard Kubernetes extension methods.

Kubernetes is composed of a control plane that is responsible for managing the totality of the cluster. It is composed of four main elements: the API Server that exposes the Kubernetes API; the etcd database, a NoSQL database that provides storage to the cluster; the controller-manager (c-m) that manages the native Kubernetes resources; and the scheduler (sched) that assigns workloads to nodes.

The rest of the cluster is composed of worker nodes. These nodes provide the processing capabilities to the cluster, and it is where the applications are deployed. There are two components that run on every worker node and that represent the agent deployed on a Worker Node: the kubelet that is responsible for running Pods and reporting the state of the node to the master and the kube-proxy (k-proxy) that enables Pod communication inside of the cluster.

It is noticeable that some efforts have been made to develop orchestration solutions apart from the leading orchestrators [21]. However, these solutions usually fail to cover the totality of the requirements presented in the flow of application deployment and cannot compete with the coverage of state-of-the-art container orchestrators. Kubernetes, being the de facto standard in the industry, becomes the go-to solution for deploying and orchestrating applications. Kubernetes is highly configurable, providing several methods for its extension without the need to patch its open-source code [35]. Furthermore, according to [15], Kubernetes is a much more extensible container orchestrator framework than other popular solutions. Therefore, the extension capabilities offered by Kubernetes make it possible to develop independent addons or plugins that aim to cover some of the features Kubernetes is missing [12,13,21,25–28], or to extend some of the functionalities of

the Kubernetes control plane elements [20,29–33,36]. These extensions, if following standard extension methods [13,27,28], can be immediately applicable to already established Kubernetes production environments, making them suitable for companies to implement. Our proposal focuses on two extension methods:

- API extensions: Kubernetes works with a set of native or built-in resources used to deploy, run and manage the required Kubernetes resources, such as, Pods, Deployments, Nodes or Services. Most of the users will have their requirements met with these. However, user-defined resources can be added to the Kubernetes API. These new resources are called Custom Resources (CRs) and are usually implemented by declaring resource types or schemas in Kubernetes through CRDs. When a new resource type is added to Kubernetes using CRDs, all of the Kubernetes tools, such as kubectl (CLI tool for Kubernetes), can be used to interact with them. From the Model-Driven Engineering (MDE) point of view, CRDs act as meta-models that define the characterization and composition rules of the different resource types, while CRs are models or instances of a certain CRD.
- Controllers: By themselves, CRs only act as a recipient or model containing certain information. As these are not built-in Kubernetes resources, the orchestrator (c-m component) does not intervene in their management. Controllers are clients of the API server and are responsible for the management of their associated CRs.

The composition of Controllers and custom resources constitutes what Kubernetes calls the operator pattern [37]. Each kind of CR has a Controller responsible for its management and Controllers follow the Kubernetes control loop (constantly watching their associated resources and accordingly acting when detecting deviations between the desired resource specification and the actual in-cluster resource state).

Figure 6 shows how the Kubernetes cluster is extended to include our proposal. The native Kubernetes control plane is divided into the control plane and the management plane. The control plane is responsible for orchestrating the new resources deployed in the cluster. The CRDs and the CRs that are instantiated based on the definition provided by the CRDs are stored inside of the etcd database.

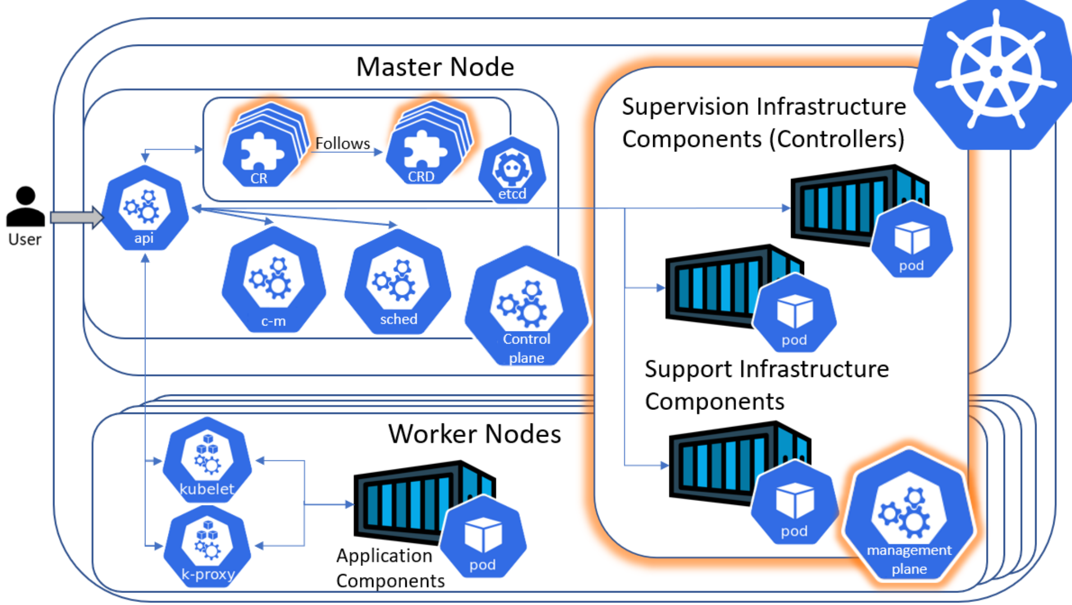

Figure 6. Kubernetes cluster with the extension elements highlighted in orange. Icons made by Pause08 www.flaticon.com (accessed on 6 June 2023).

The newly defined management plane is composed of the Supervision Infrastructure Components and the Support Infrastructure Components. The former are deployed in the Master Node. In contrast, the latter are not bound to the Master Node, and they can be deployed in certain Worker Nodes according to different criteria. For example, location-awareness or latency-awareness of application components require the use of certain Infrastructure Services or deployment in Worker Nodes with certain capabilities. Therefore, the management plane is extended to cover both Worker and Master nodes.

3.4. Case Study

This section presents: (a) the testbed on which the case study was deployed; (b) the HAMS designed for the case study; (c) the Support Infrastructure Components available for the applications; and (d) the applications designed to validate the proposal.

3.4.1. Testbed

Industry 4.0 consists of the integration of multiple technologies that can be deployed into several control layers. Thus, the specific multi-layer approach considered in this case study is aligned with OpenFog and comprises the IoT, Fog and Cloud layers presented in Figure 7.

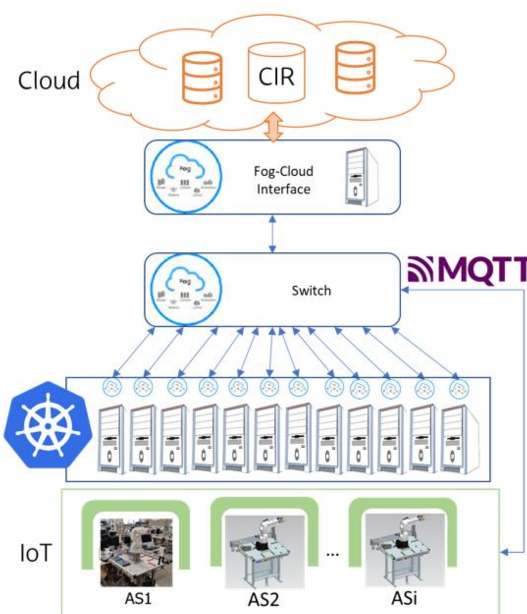

Figure 7. Testbed available to deploy the case study. The elements of the IoT layer are represented in green, the elements of the Fog layer in blue and the elements of the Cloud layer in orange.

In this testbed, the data production is at the IoT layer, composed of a manufacturing cell that performs the assembly operations of a set of 3D printed parts emulating the shaft of a stepper motor. The assembly cell comprises a KUKA KR3 R540 robot [38], managed by a Siemens ET 200SP Open Controller [39], which embeds a virtualized PLC and Windows 10 operating system. To have greater flexibility in performing the tests, the testbed also has Digital Twins (DTs) to simulate additional manufacturing cells. These DTs are modeled and simulated using the software-in-the-loop approach with Tecnomatix Process Simulate 16.0.1 [40] and PLCSIM Advanced V3.0 [41] tools. All the equipment in this testbed was sourced from the respective local distributors in Spain.

After each batch process, the production data are sent to the Fog layer through the Fog-IoT Message Hub. The Fog layer receives the data produced by the IoT layer and turns the data collected into insights with the aim of taking decisions that improve the efficiency and productivity of the production processes. The elements that form the testbed in the Fog layer can be classified as follows:

- The Fog Computing cluster is based on 12 DELL Optiplex 780 computers running Ubuntu 20.04 and connected through an unmanaged D-Link DGS-1024D gigabit switch. One of the computers acts as a Master Node while the rest of the computers act as worker nodes. Each one of the Fog Nodes runs K3s [42], a lightweight, certified Kubernetes distribution specifically designed for IoT applications.
- A DELL Optiplex 7010 is used as a gateway between the Fog Cluster and the Cloud, implementing the Fog-Cloud Interface of our proposed architecture. This enables the Fog layer to reach the Container Image Registry when needed.

3.4.2. Designed HAMS

The first step to implementing the platform is to define the HAMS. To do so, the organizational and business needs must be analyzed. As stated in the previous subsection, the testbed is composed of industrial robots and their digital twins, considered manufacturing assets. That sets the first level of the HAMS (Asset). Assets are elements that execute unitary operations on the products to add value during the whole Assembly Line processing. Each Asset has a set of DAGs assigned to it, which represent the logical applications that process the information related to each Asset.

From an organizational point of view, it is convenient to group assets that interact in the same assembly line. Therefore, the second level (Assembly Line) is identified as a composition of different Assets. The sum of these two organizational levels sets the number of levels presented in the HAMS (i.e., N = 2).

3.4.3. Support Infrastructure Components

As previously explained, the services offered by Support Infrastructure Components do not have any specific use on their own and cannot accomplish any business need. However, they provide infrastructure services to the application components that do achieve business goals. Therefore, the design of applications is dependent on the Support Infrastructure Components available and the services they offer. For this case study, five components have been considered:

- MQTT broker: It runs the Fog-IoT Message Hub that acts as a link between the IoT and Fog layers. As its name suggests, it uses MQTT, a lightweight message protocol used in IoT applications, which is based on a publish-subscribe model.
- Kafka: The Intra-Fog Message Hub is implemented through Kafka. It is an event streaming platform based on a publish-subscribe model. It allows and enables the communication of the application components that run on the cluster.
- eXist-DB: A NoSQL database designed to store XML data in a native way.
- InfluxDB: A NoSQL database designed to store time series data.
- Grafana: A tool for monitoring and analyzing data from different databases. It allows visualizing the data processed and stored by the applications.

3.4.4. Designed Applications (DAGs)

The objective of these applications is to collect and process the data obtained from the IoT layer. Specifically, we selected the Overall Equipment Effectiveness (herein OEE) as the Key Performance Indicator (KPI) to measure the productivity of the assets. The OEE is recognized as one of the most insightful KPIs to assess the overall productivity of a given manufacturing asset [43]. Two applications are designed, developed and deployed per asset:

1. Acquisition: This application obtains the data produced by the asset and stores it on a NoSQL database. Its logic is divided into two components: one that subscribes to the MQTT broker and obtains the manufacturing data (C_1), and the other that stores that data inside the eXist database (C_2). Note that the MQTT broker represents the Fog-IoT Message Hub.
2. Processing: It is a data processing application that is composed of three components as follows: the first component (C_1) reads the previously stored data from the eXist database and sends it to the second component; then, the second component (Process)

is a permanent component that processes the data and extracts the OEE from it; finally, the OEE is sent to the third component (C_3) that stores it in a InfluxDB NoSQL database.

To illustrate the design of these applications, Figure 8 represents the two DAGs for the Asset *Assembly Robot_1* in the application structure designed. To help understand the relations between Application Components and Support Infrastructure Components, a bottom layer has been added to show Support Infrastructure Components and their use by the Application Services.

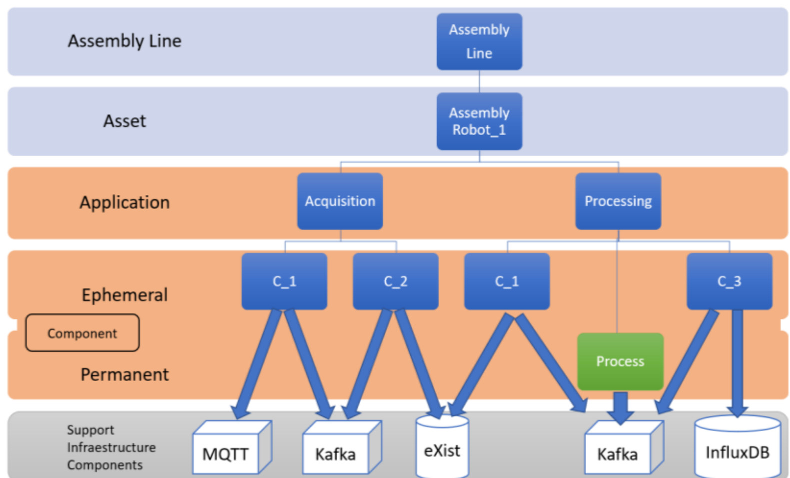

Figure 8. Applications are designed per asset that forms the Assembly Lines. Permanent component C_2 in the Processing application is highlighted in green.

These applications can be also represented as DAGs, as shown in Figure 9. In this view, the usage of Support Infrastructure Components is inherent to the programming of the individual application components and, thus, is not represented. Figure 9 focuses on the communication between components, which is based on the Intra-Fog Message Hub implemented using Kafka. Therefore, the links between components represent the Kafka topics where data are transmitted.

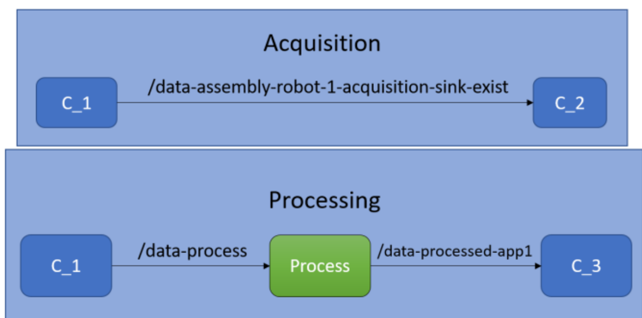

Figure 9. *Assembly Robot_1* applications as DAGs.

The complete case study, with the four applications designed for the two Assets, is represented in the application structure in Figure 10. The addition of the second Asset, *Assembly Robot_2*, is an exemplification of horizontal scaling applied to an already-existing structure. As the *Process* permanent component offers a service used in both applications, it is represented as a component of both applications.

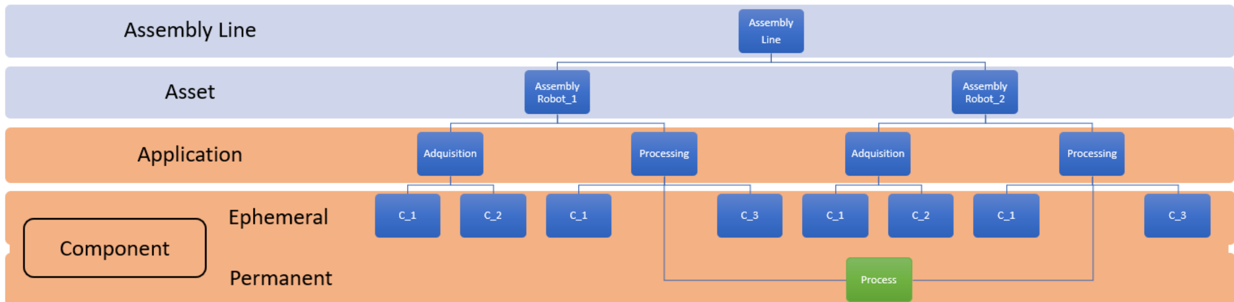

Figure 10. Additional asset added in the assembly line. The component *Process* is used in both Processing applications.

4. Results

This section explains how the proposed architecture and application management hierarchy are implemented in Kubernetes utilizing the materials and methods presented in the previous section, resulting in an application-aware orchestration platform, which is also OpenFog compliant.

The set of components detailed in this section comprise the Supervision Infrastructure Components of the platform. They manage the resources that are dynamically created and decommissioned in the system. Kubernetes states that "*Whether your workload is a single component or several that work together, on Kubernetes you run it inside a set of Pods*" [34]. These Pods are then managed by the so-called workload resources that manage sets of Pods. These workload resources establish controllers and behaviors additional to those provided by Kubernetes that can be implemented by using CRDs. Therefore, to include Kubernetes at all levels of the application structure, i.e., every level of the HAMS as well as the two levels identified for DAGs, we propose to implement a CRD, a controller and the necessary access control files for each of the levels.

The CRD is a generic definition or meta model of a type of resource. It is written in YAML (a subset of JSON) and its structure is pre-established by Kubernetes, divided into different sections: apiVersion, Kind, metadata, spec and status. The most important section is the spec field (see Figure 11). It defines the structure that every instantiated CR must follow based on its related CRD. It is divided into two subsections, namely spec and status. The spec subsection is generated based on the openAPIv3 specification. It is separated into optional and required fields, which are checked by Kubernetes against the CRD when a related CR is instantiated. The status subsection is used to store state information of the resources so the controllers can compare the actual status of the deployed object and the desired state stored as a specification in the spec section. The status subsection must be enabled in the subresources section. In addition, AdditionalPrinterColumns are used to determine the field parameters when utilizing the CLI to interact with the object, providing additional information about the state of the resources to the cluster administrator. Additionally, it is compulsory to define the names used to interact with the instantiated CRs by the users from the CLI or by Kubernetes itself, in the names subsection.

Each CRD is accompanied by a controller, implementing the operator concept and achieving a fully declarative API. Each controller, acting as a Supervision Infrastructure Component, has been designed to manage the resources of its level during their lifecycle and to create and decommission the resources of its lower level. The state machine in Figure 12 depicts the lifecycle proposed for a controller at any application structure level. When the controller is initiated, it transitions to the Starting state in which the controller reads the cluster configuration and connects to the Kubernetes API as a client. When the controller is started, i.e., the client is created, it transitions to the Running state. Here, the controller activates a watcher that must be aware of the resources of its level and the events they generate to process them. Events represent the change in the state of a certain resource inside the cluster. Our controllers distinguish three Kubernetes event types: added,

modified and deleted. The watcher has been designed to process them through several functions (right part of Figure 12):

- *Added events* represent events related to a resource that has been instantiated for the first time in the system. The resource is passed to the *Create Lower Resources()* function, which instantiates the individual resources that compose the newly instantiated CR.
- *Modified events* reflect that the related resource has been somehow modified. Affected resources are passed to the *Reconcile Spec Status()* function that reads the current object status, compares it to the desired spec in the CR body and accordingly acts.
- *Deleted events* are raised by Deleted resources. The resource is passed to the *Delete Lower Resources()* function that decommissions all the lower-level resources that compose a given CR.

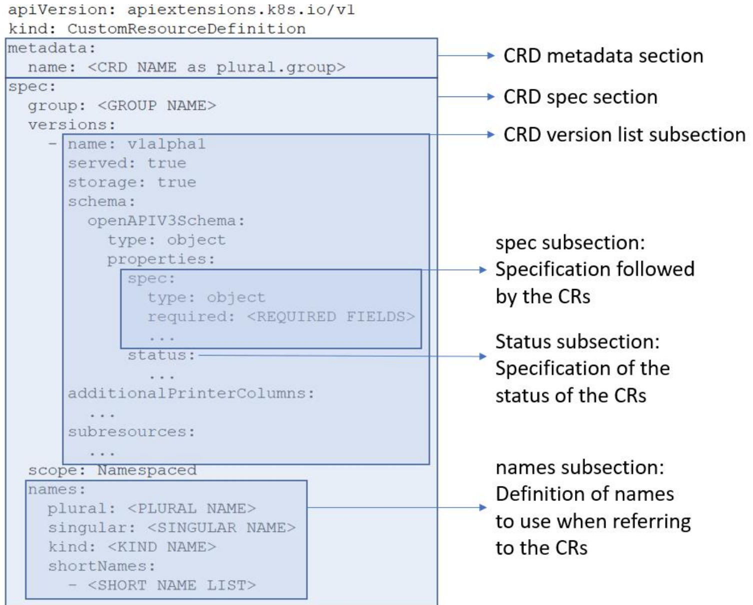

Figure 11. CRD body example.

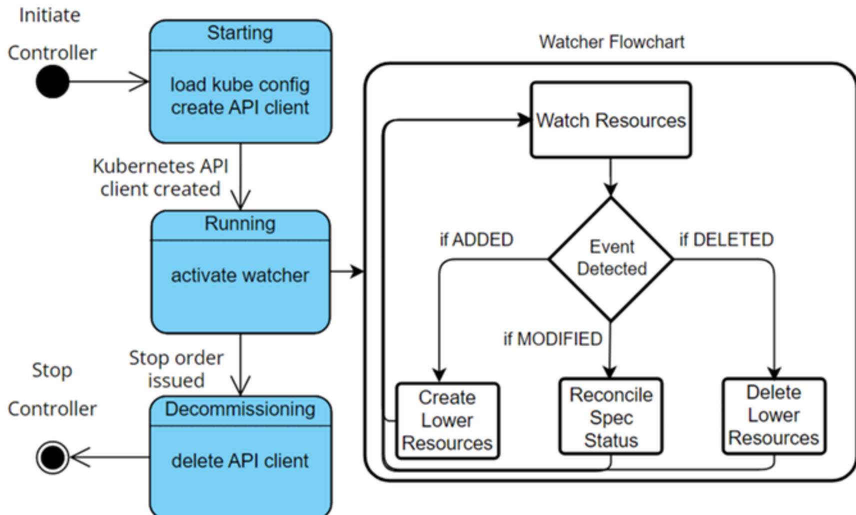

Figure 12. Default state machine of a given level controller.

Controllers are designed to stay in the running state as long as the platform is deployed and running. When the controller is decommissioned, it transitions from the running state to the Decommissioning state. The watcher is stopped, the Kubernetes client is deleted and the controller is stopped.

Kubernetes bases the access to its API resources on a Role-Based Access Control method (RBAC). In our case, we utilize the Kubernetes ClusterRole resource to provide cluster-wide access to a controller for a given resource. This way, there is a certain separation of concerns when it comes to the responsibilities of each controller. This ensures that no resource is modified by any process that should not alter it and that no process has access to information it should not handle. For example, in Appendix A, Figure A1a shows the body of a ClusterRole resource for a controller. Here, the resources to be accessed are defined and the <VERB_LIST> specifies the HTTP verbs allowed for the HTTP calls made by the user that has this ClusterRole assigned. Furthermore, Kubernetes requires these roles to be explicitly assigned to a ServiceAccount. This is performed by using ClusterRoleBinding resources; Figure A1b shows the body of an example ClusterRoleBinding. The roleRef is assigned to the defined subjects; in this case, the developed SeviceAccounts. The platform is distributed as a Kubernetes operator containing the YAML files required to deploy the DAG levels. In addition, an Application Structure Generator is provided as a Python script for users to determine the application structure desired. This tool automatically creates all the necessary HAMS implementation files to create a functional application structure ready to be deployed on Kubernetes. The generator also parameterizes the Application Controller if there is a level above it. Each level consists of five YAML files to be deployed in Kubernetes: the CRD, the controller Deployment and three YAML files to enable RBAC (a ServiceAccount, a ClusterRole and a ClusterRoleBinding). The operator code and the generator files can be found in the Data Availability section.

Controllers can be implemented in any programming language that can communicate with the Kubernetes API. We selected Python as the programming language due to its extensive library pool and ease of implementation. The controller code must be packaged in a container image and must run inside a Kubernetes Pod declared in a Kubernetes Deployment. The HAMS controllers are implemented based on the same generic controller container image that is parameterized by the Application Structure Generator.

The following subsections describe how these extension methods are applied to include the different levels of the proposed application structure into the native Kubernetes to obtain an application-aware platform.

4.1. DAG Levels

As explained in previous sections, applications are often represented as graphs, specifically, DAGs. Our application structure considers the application as a DAG and distinguishes two levels: Application and Component. Every HAMS, let N = 0 or N = 10, implements these lower levels as the representation of DAGs. The Application and Component levels serve as a bridge between the conceptual representation of the real world presented in the HAMS and the deployment of the individual microservices after the processing of the components. Therefore, as these two levels are inherent to our proposal, their implementation cannot be modified by the users.

4.1.1. Component Level

The lower level refers to the Component resources that implement the business logic. In Appendix A, Figure A2 presents how the generic CRD (see Figure 11) is particularized for the component resource. The spec subsection includes four required fields:

- name: Name of the component resource as seen in Kubernetes. It must be unique in the system; therefore, ephemeral components are named as Application.name-Component.name. On the contrary, permanent components, as they take part in numerous applications, are given a name equal to Component.name. Permanent components will be unique in the system and, thus, no further identification is required.

- image: Name of the image to be pulled by Kubernetes. The container image includes the necessary code to run the component's functionality and is accordingly parameterized to interpret the rest of the component fields.
- flowConfig: This parameter holds the information related to the flow of the DAG relative to the component. It is subdivided into two parameters, namely previous and next:
    - previous: An array of a duple of the name of the previous components in the flow and the topic (IFMHtopic, Intra Fog Message Hub topic) on which the component will be offering its service.
    - next: An array of a duple of the name in the flow and the topic (IFMHtopic) of the next components on which the component will be publishing to reach the next components.

The rest of the fields are considered optional. Therefore, they are not required when instantiating a component:

- customization: This field allows for the customization of a component with any additional information that might be relevant to its runtime environment, passed as Strings defined by the component developer. It is useful when the component developer sets additional requirements not part of the functioning of the platform but needed by the component code.
- permanent: The persistence of the component is reflected in this field. If a component is permanent, this field is set to True. The Component Controller will act differently depending on the persistence of the component.
- permanentCM: Related to the previous field, it indicates the name of the ConfigMap (Kubernetes object used as key-value storage) the permanent component will use as its configuration file. Ephemeral components are customized at the time of deployment using the previously explained parameters, as this type of component does not form part of different applications during their lifecycle. However, the configuration of permanent components is variable during runtime, and it must be updated when new applications request their services. Therefore, ConfigMaps are used to dynamically change the configuration of a permanent component without the need to redeploy it (i.e., to consider the permanent component as a node in different DAGs at the same time). Specifically, a ConfigMap is linked to a Volume (a directory containing data in the Kubernetes cluster for Pods to consume) so when the ConfigMap is updated, the information is ready for the container to access in the Volume. This process is controlled by Kubernetes and is independent of both the platform and the user.

The status subsection of the Component CRD Is used to track the actual state of the component resources in the cluster. To do so, two fields are used:

- replicas: It represents the number of available replicas in a running state in the cluster.
- situation: It is a parameter that reflects the current status of the component (Deploying or Running). The initial situation is Deploying, but when all the desired replicas are running, the component controller updates it to Running.

4.1.2. Application Level

The upper level refers to the DAG elements. Figure A3, shown in Appendix A, shows how the generic CRD (see Figure 11) is customized for the application resources. In contrast to components, the fields that define an application's spec are all required:

- name: Name of the application resource. It must be unique in the system and, thus, they are named as a composition of the upper resource name and the application name.
- components: As applications are composed of a set of components, this field is an array of component definitions that follow the specification shown in Figure A2.
- replicas: The purpose of this field is to specify the number of active replicas desired for a given application. This is then translated into as many components as the number of replicas desired.

- deploy: This parameter is used to indicate whether to deploy the application as soon as the resource is declared on Kubernetes or whether to store the application resource defined and decide afterward to deploy the application. It is useful to define several applications in the cluster that are expected to be run afterward.

As for the status subsection, it is used to track the actual state of the application resources and is divided into several fields as follows:

- replicas: This field is used to track the number of application replicas that are running at a given time.
- components: Used to track the status of the different components that form the application. For each component, its name and its situation, specifically, the status.situation of the component, are stored. The Application controller reacts to the change in its component's status and accordingly updates the whole application's status.
- ready: It is an enumerated parameter that stores the actual application state (deploying or running). When all the components that form an application transition to a running state, this parameter is changed from deploying to running.

The sequence diagram in Figure 13 illustrates the high-level interactions between the two DAG level controllers (management plane) and the Kubernetes control plane elements. The initial state is a Kubernetes cluster that has been extended by deploying the Application and Component CRDs, the ServiceAccounts, ClusterRoles and ClusterRoleBindings and the Application and Component Controllers (which are already in their Running states). Our controllers are highlighted in blue to distinguish them from the Kubernetes control plane elements. The overall sequence diagram can be divided into three sections that are coherent with the functions presented in the watcher flowchart of the controller:

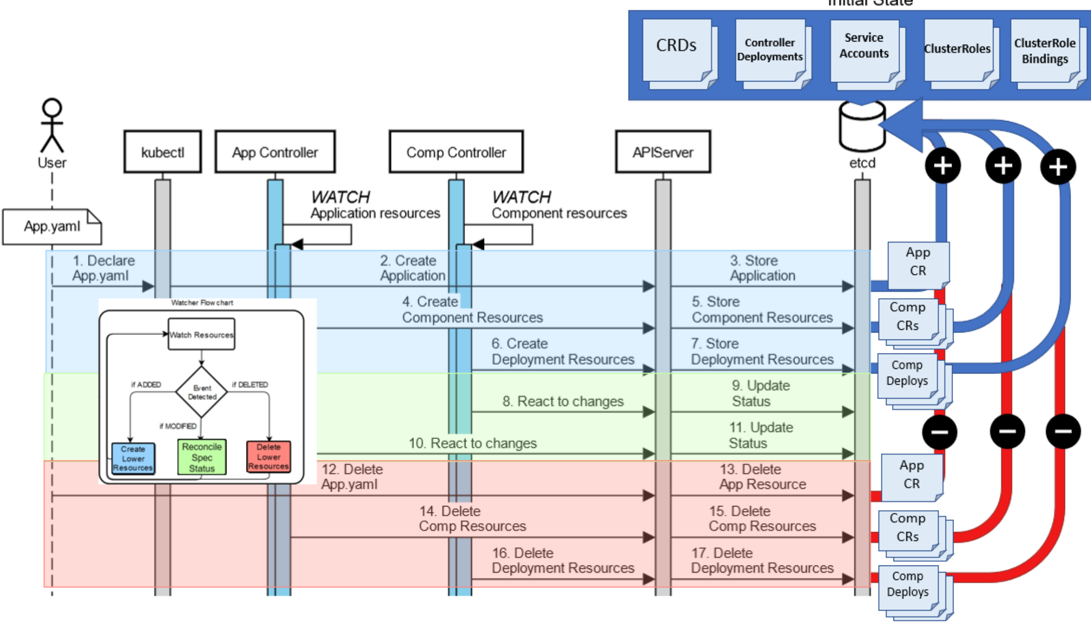

Figure 13. Sequence diagram of the high-level interactions between the Application (App) and Component (Comp) controllers with the Kubernetes control plane. The creation of resources is highlighted in a blue section, the conciliation of the desired spec and the current status is highlighted in green and the deletion of resources is highlighted in red. The right side represents the addition (in blue and with a '+' sign) and subtraction (in red and with a '−' sign) of resources that dynamically occur over the etcd.

- The creation of resources is initiated by an external user. An application CR is created following its corresponding CRD metamodel, which results in a YAML file. Then, the user declares the application in the cluster by posting the YAML file using the kubectl CLI, and it is stored in the etcd. Then, the Application Controller watcher detects the

creation of the application resource and creates the Component CRs. Accordingly, the Component Controller watcher detects the creation of the components and creates the Deployment resources.
- The controllers then seek the conciliation of the desired spec definition and the current status of the resources. After the Deployments have been instantiated, the Component Controller reacts to the changes that reflect the status of the components and updates them in the database. The Application Controller detects changes in its resources and updates its status in the etcd.
- The deletion of the resource follows a similar pattern to the creation. First, the user requests the deletion of the resource (an application). Then, the Application Controller detects the deletion of the resource and initiates a cascade deletion of all its components. Finally, the Component Controller deletes the Deployment Resources.

It should be noted that the creation of Pods and the instantiation of containers is a responsibility of Kubernetes; i.e., DAG level controllers do not have to deal with the Deployment resources. The same applies to the deletion of the resources: DAG level Controllers accordingly delete Deployments and Kubernetes acts.

4.2. User-Defined N HAMS Levels

As mentioned above, the levels that compose the HAMS are implemented by a CRD, a controller, a ServiceAccount, a ClusterRole and a ClusterRoleBinding. The decision of the number of levels that form the HAMS must be made prior to the extension of Kubernetes and, thus, prior to the deployment of the platform. The number of levels (N) must be indicated, and the names of the spec section of each CRD must be specified. This information is used by the Application Structure Generator provided by the operator to generate the YAML files relative to each level. Figure A4 shows how the generic CRD (see Figure 11) is customized for the first level of the HAMS, that is, the level above the DAG levels.

The spec subsection of these resources is based on three parameters: one for their name, another one for its list of applications and a third one to decide whether to deploy the resource at the same time it is instantiated in the cluster. The status subsection of the resources consists of an array to store the state of the applications by their name and situation, as well as the ready field. This latter represents whether the applications related to the resource have transitioned to the running state and, thus, the resource's ready field can be updated from Deploying to Running. The rest of the CRDs related to the additional HAMS level are generated using the same nesting logic shown in Figure A5.

The spec subsection of these CRDs is a nested list of arrays of the lower resources. Thus, the CRD generation begins at level 2 and continues to the Nth level. Furthermore, their name must be specified as well as whether to deploy it. The status subsection, as for the previous case, reflects the name and situation of the lower resources as well as the state of the actual level resources in the ready field (Deploying or Running).

The controllers for each of the levels utilize the same controller image, accordingly parameterized with the names of their upper, current and lower resources. For the uppermost level, the upper level is indicated as System, and thus, the controller interprets that it must not search for upper resources. The logic of these controllers is the same as the logic represented in Figure 12. The sequence diagram in Figure 14 illustrates the high-level interactions between the controllers for the HAMS levels, the Application and Component Controllers and the Kubernetes control plane elements.

The initial state is equal to the one shown in Figure 13, although extended with the files necessary for the HAMS levels. All of the application structure level controllers activate watchers that are watching events in the system related to their managed resources. The sequence starts with a user that declares the Level N Resource Definition.yaml file. The HAMS level N Controller watcher detects the creation of the Level N Resource and issues the creation of the Level N-1 Resources that compose the Level N Resource. Then, for every controller related to the levels $i \in [2, N-1]$, the watchers detect the creation of their respective resources and issue the creation of their lower-level resources. When the HAMS

Level 1 Resources are created, the HAMS Level 1 controller watcher detects the event and issues the creation of application resources, starting the sequence described in Figure 13.

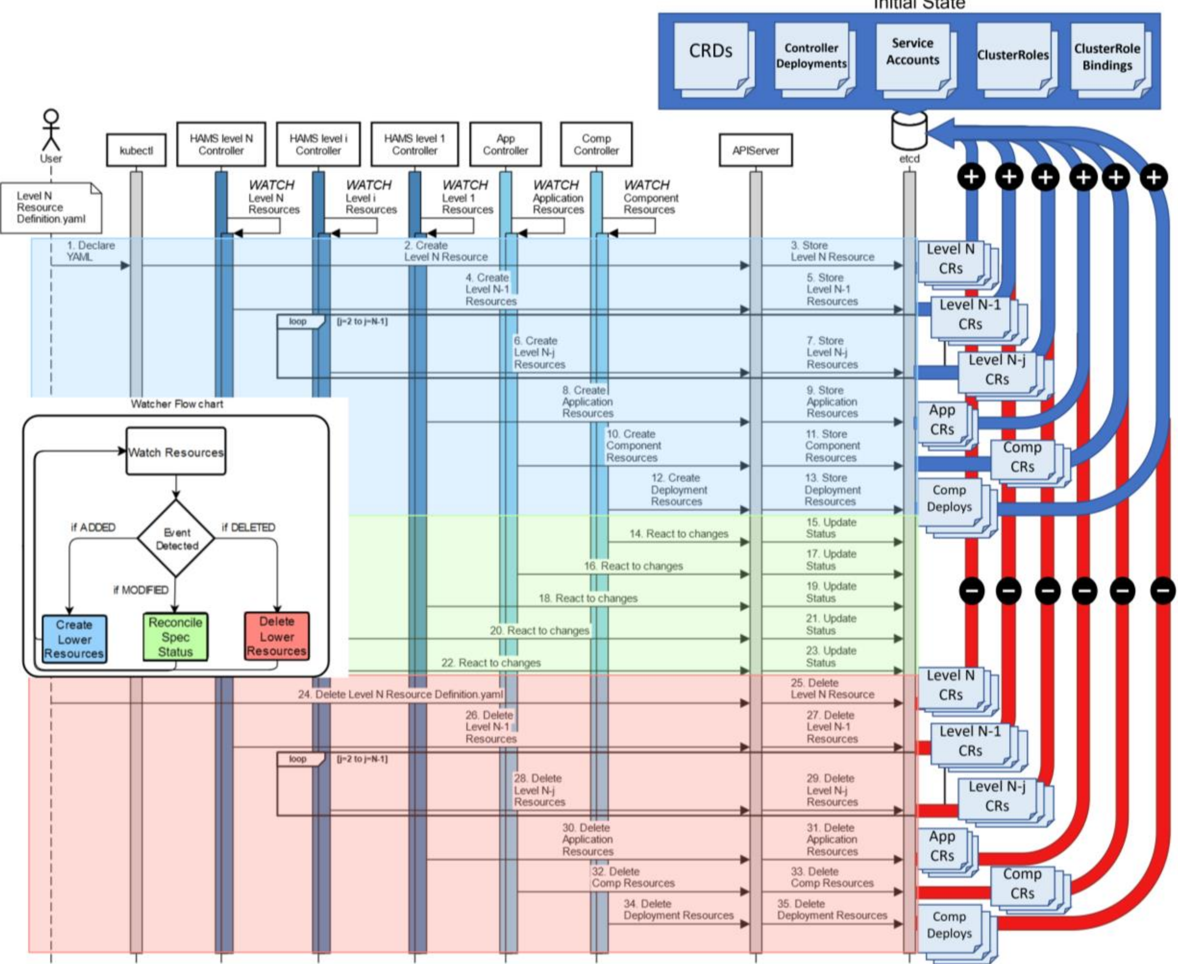

Figure 14. Sequence diagram of the high-level interactions of the complete application structure and the Kubernetes Control Plane. The creation of resources is highlighted in the blue section, the conciliation of the desired spec and the current status is highlighted in green and the deletion of resources is highlighted in red. The right side represents the addition (in blue and with a '+' sign) and subtraction (in red and with a '−' sign) of resources that dynamically occur over the etcd.

5. Discussion

This section illustrates how to utilize the platform developed and how to deploy the designed applications on the platform built on top of Kubernetes. The applicability of this method is not fixed to this example and can be adapted to a user-defined HAMS in any domain. To help visualize the deployment of the platform over a vanilla Kubernetes cluster, we utilize screenshots from the Kubernetes dashboard.

First, the application structure is defined (explained in Section 3.4.2), based on the structural needs of the user: a two-level HAMS, where assembly lines represent asset groups and DAGs are defined for each asset. Then, the Application Structure Generator provided by the operator is used to create all the YAML files relative to each of the levels. With the YAML files developed, the application structure is deployed on the cluster (i.e., the controllers, the CRDs and their respective RBAC files, enabling the management of said resources). Figure 15 shows, as seen from the Kubernetes dashboard, that the CRDs related to our application structure are present and deployed and that there is a running Deployment for every controller related to each level.

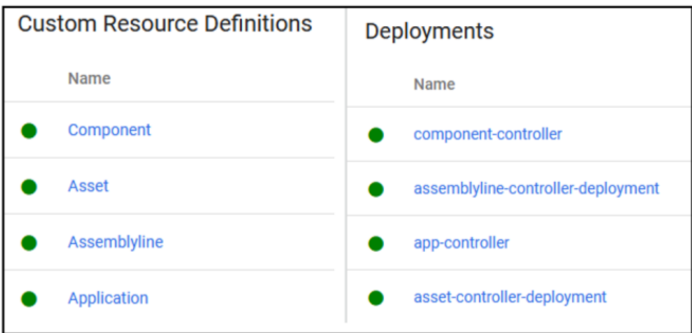

Figure 15. CRDs and Deployments related to the deployment of the platform.

When the Applications are designed and the Components are developed (Section 3.4.4), the construction of the YAML file containing all of the Assembly Line's information is started. This custom resource must follow the CRD definition of the Assembly Line resources, as structured in Figure A5. In this case, the Assembly Line contains two Assets, while each of the Assets contains two Applications (see Figure 10). The Support Infrastructure Components are identified (see Figure 8) and deployed in the cluster.

Once constructed, the AssemblyLine.yaml file is posted in Kubernetes through the kubectl CLI. According to Figure 14, this action starts the deployment process for the subsequent resources. Once the AssemblyLine resource is created, two Asset resources are created and four application resources are created. Lastly, nine component resources are created, one of which is a permanent component while the rest are ephemeral components. Figure 16 shows these elements as seen from the dashboard, where the user can interact with them and with the Pods through which they are deployed, analyze resource consumption, check logs, etc.

Metadata
Name
assemblylines.ehu.gcis.org
Name
assemblyline

Metadata
Name
assets.ehu.gcis.org
Name
assemblyline-assembly-robot-1
assemblyline-assembly-robot-2

Metadata
Name
components.ehu.gcis.org
Name
assembly-robot-1-acquisition-source-mqtt-kafka
assembly-robot-1-acquisition-sink-exist
assembly-robot-1-processing-exist
process
assembly-robot-1-processing-influx
assembly-robot-2-acquisition-mqtt-kafka
assembly-robot-2-acquisition-sink-exist
assembly-robot-2-processing-exist
assembly-robot-2-processing-influx

Metadata
Name
applications.ehu.gcis.org
Name
assembly-robot-1-acquisition
assembly-robot-1-processing
assembly-robot-2-acquisition
assembly-robot-2-processing

Figure 16. Resources deployed on the cluster as seen from the Kubernetes dashboard.

As an example, Figure 17 shows the events related to the Asset Assembly Robot_1. It should be noted how, first, the Asset is detected as successfully created and then, as the Applications are instantiated and transitioned to the Running state, the Asset itself is tagged as Running.

Metadata

Name
assemblyline-assembly-robot-1

Events

Name	Reason	Message
assemblyline-assembly-robot-1-Running-1gdqm	Running	All lower resources successfully deployed.
assemblyline-assembly-robot-1-Created-4v45s	Created	asset successfully created.
assemblyline-assembly-robot-1-Created-81m95	Created	application successfully created by asset.
assemblyline-assembly-robot-1-Created-yvqdy	Created	application successfully created by asset.

Figure 17. Events related to the Asset Assembly Robot_1.

Figure 18 reflects the events relative to the Process permanent component. It should be noted that the permanent component's ConfigMap is created with the first application that instantiates the component. Afterward, the second application that uses the permanent component modifies the existing ConfigMap adding its information, thus, making it available to the component.

Metadata

Name
process

Events

Name	Reason	Message
process-deployed-3qqto	Running	Component successfully deployed.
process-created-60d8c	Created	Component successfully created.
process-modified-b4rdi	Modified	Permanent component added to assembly-robot-2-processing application.
process-Created-34jbi	Created	Permanent component's ConfigMap created. The related application is assembly-robot-1-processing.

Figure 18. Events related to the Process permanent component.

To reflect the operation of the applications as they process the OEE of the two Assets, Figure 19 shows how the OEE is calculated through time. To that end, the Grafana Support Infrastructure Component is used. It ingests the data left by the Processing applications at the InfluxDB Support Infrastructure Component. To help understand the changes that take place in the cluster when the resources are created, two videos are provided as Supplementary Materials.

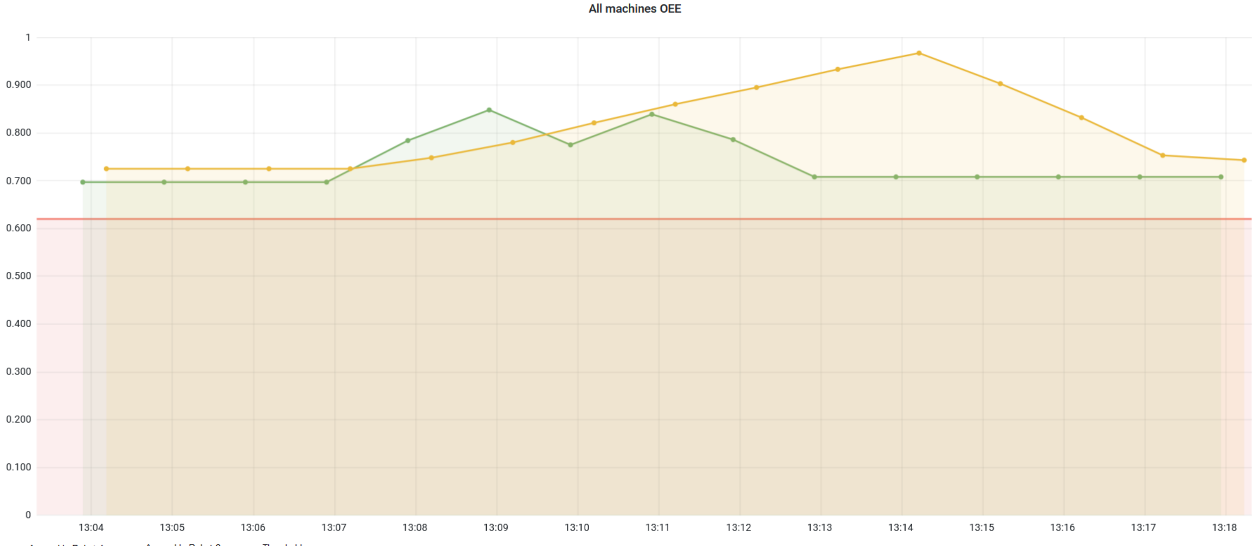

Figure 19. OEE calculation of the two Assets through time. The Vertical axis represents the OEE value and the horizontal axis represents its calculation time. The green and yellow points represent the computed OEE for Assembly Robot_1 and Assembly Robot_2, respectively. The orange points depict the OEE threshold under which the machine performance is not acceptable.

6. Conclusions

Fog Computing presents a suitable environment for offering Cloud-like services closer to the sources of data. However, to the authors' knowledge, container orchestrators are not aware of the collective representation of their components as applications, nor do they support their logical grouping in a hierarchy based on the operating model of a system or organization.

This work contributes with an application-aware platform that enables the design of a HAMS the elements of which have applications designed as DAGs associated. This approach enables the consideration of abstract concepts from the organizational or business structure at runtime. Thus, the deployment of microservice-based Fog Applications may include these abstract representations utilized in the HAMS definition. The approach described in this work implements the presented architecture over Kubernetes, utilizing its standardized extension methods.

The Supervision Infrastructure Services deployed as Kubernetes controllers enable the separation of concerns between the different phases and actors found when designing, developing and deploying Fog Applications. It abstracts the application designer from the usage of the container orchestration tool and from the development of Support Infrastructure Components and Application Components. Furthermore, the usage of an extended Kubernetes platform itself allows users to decouple the deployment of the application structure elements from the runtime management.

The intelligence of the controllers makes it possible to provide scalability during runtime to consider additional assets or applications in the platform or to include additional abstractions as levels in the HAMS. Being implemented based on standard Kubernetes extension methods, the platform could be extended with other available tools or operators to cover some requirements that the proposal might not cover.

However, the proposal has some disadvantages, as in its current state it only allows the communication of the microservices through a publish-subscribe mechanism that requires the programmers of the components to implement that functionality and make it parameterizable with the information provided in the component resource. Furthermore, the proposal requires users to develop quite complex YAML files that could be automated in further work using MDE. Additionally, although the HAMS provides some flexibility, it cannot be modified during runtime and only one Application Structure might be deployed at a time.

Future work is also aimed at exploring the performance limitations of the proposal through quantitative analysis and stress tests, searching for the limits of the controllers developed and the application structure as a whole. In those cases, the autoscaling feature of Kubernetes could be explored to scale controllers as needed. Scalability should also be introduced in Application Components as the current approach only considers static scalability or replicas.

Future work might also consider other different routes. The inclusion of more complex DAGs and the coexistence of several simultaneous HAMS in the system should be explored. It should be considered that every component might offer more than one service to different entities. Support for direct synchronous communication between components could be introduced as an alternative to the presented topic-based publish-subscribe communication paradigm. The design and development of the HAMS and its elements could be performed through a more user-friendly editor. Node restrictions and awareness could be improved when deploying Fog Components.

Supplementary Materials: The following videos serve as supplementary material to the article: https://www.youtube.com/playlist?list=PLs6bFF_iqW3H5YDCk599_OFbxnpPHL4gr (accessed on 12 June 2023).

Author Contributions: Conceptualization, J.C., O.C. and A.A.; methodology, J.C., E.H., O.C. and A.A.; software, J.C. and E.H.; validation, J.C. and E.H.; formal analysis, J.C., O.C. and A.A.; investigation, J.C., E.H., O.C. and A.A.; resources, O.C., A.A. and F.P.; data curation, J.C. and E.H.; writing—original draft preparation, J.C., O.C. and A.A.; writing—review and editing, J.C., E.H., O.C., A.A. and F.P.; visualization, J.C., O.C. and A.A.; supervision, O.C. and A.A.; project administration, O.C. and F.P.; funding acquisition, O.C., A.A. and F.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the project PES18/48 funded by the University of the Basque Country (UPV/EHU) and by the PhD fellowship granted under the frame of the PIF 2022 call funded by the University of the Basque Country (UPV/EHU), grant number PIF22/188.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The files related to the operator and the Application Structure Generator can be found in the following GitHub repository: https://github.com/JulenCuadra/AppAwarePlatform (accessed on 12 June 2023).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in the manuscript:

Acronym	Definition
ACOA	Application-Centric Orchestration Architecture
AL	Assembly Line
API	Application Programming Interface
AR	Assembly Robot
CIR	Container Image Registry
CLI	Command Line Interface
CR	Custom Resource
CRD	Custom Resource Definition
DB	Data Base
DT	Digital Twin
DG	Directed Graph
DAG	Directed Acyclic Graph
HAMS	Hierarchical Application Management Structure
HTTP	Hypertext Transfer Protocol
IFMHtopic	Intra Fog Message Hub topic

IoT	Internet of Things
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MAPE	Monitor, Analyze, Plan and Execute
MDE	Model-Driven Engineering
MQTT	Message Queuing Telemetry Transport
NoSQL	No Structured Query Language
OAM	Open Application Model
OEE	Overall Equipment Effectiveness
PaaS	Platform as a Service
PLC	Programmable Logic Controller
RBAC	Role Based Access Control
RPC	Remote Procedure Call
TOSCA	Topology and Orchestration Specification for Cloud Applications
TR	Transport Robot
YAML	YAML Ain't Markup Language/Yet Another Markup Language

Appendix A

This appendix contains figures used to support the text of the article.

Appendix A.1. ClusterRole and ClusterRole-Binding Resources

Figure A1 presents the files to manage the RBAC for users.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: <CLUSTER_ROLE_NAME>
rules:
- apiGroups: <GROUP of the resource to be accessed>
  resources: <RESOURCE_LIST>
  verbs: <VERB_LIST>
...
```

(**a**)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <ClusterRoleBinding_NAME>
subjects:
- kind: User
  name: <USER_NAME>
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: <ClusterRole_NAME>
  apiGroup: rbac.authorization.k8s.io
```

(**b**)

Figure A1. Role-Based Access Control: (**a**) example of a ClusterRole body and (**b**) example of a ClusterRole-Binding body.

Appendix A.2. CRD Parametrizations

Figures A2–A5 detail how the generic CRD is customized for several resource types.

Spec subsection field	Field Type	Required
name	String	Yes
image	String	Yes
flowConfig	Object	Yes
previous	Array of Objects	Yes
name	String	Yes
IFMHtopic	String	Yes
next	Array of Objects	Yes
name	String	Yes
IFMHtopic	String	Yes
customization	Array of Strings	No
permanent	Boolean	No
permanentCM	String	No

Status subsection field	Field Type
replicas	Integer
situation	String

Names subsection field	Field Type
group	String
plural	components
singular	component
kind	Component
shortname	comp

Figure A2. Component CRD parameterization.

Spec subsection field	Field Type	Required
name	String	Yes
components	Array of Components	Yes
replicas	Integer	Yes
deploy	Boolean	Yes

Names subsection field	Field Type
group	String
plural	applications
singular	application
kind	Application
shortname	app

Status subsection field	Field Type
replicas	Integer
components	Array of objects
name	String
situation	String
ready	String

Figure A3. Application CRD parameterization.

Spec subsection field	Field Type	Required
name	String	Yes
applications	Array of Applications	Yes
deploy	Boolean	Yes

Status subsection field	Field Type
applications	Array of objects
name	String
situation	String
ready	String

Names subsection field	Field Type
group	String
plural	String
singular	String
kind	String
shortname	String

Figure A4. CRD parameterization of the first HAMS level.

	Spec subsection field	Field Type	Required
	name	String	Yes
N	<current level – 1 resources>	Array of <current level – 1 resources>	Yes
...	...	...	Yes
1	applications	Array of Applications	Yes
	deploy	Boolean	Yes

Status subsection field	Field Type
<current level – 1 resources>	Array of objects
name	String
situation	String
ready	String

Names subsection field	Field Type
group	String
plural	String
singular	String
kind	String
shortname	String

Figure A5. Generic CRD parameterization of the levels 2 to N. The red text represents the nested resource specification as the N levels grow.

References

1. Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Commun. Surv. Tutor.* **2015**, *17*, 2347–2376. [CrossRef]
2. Dastjerdi, A.V.; Buyya, R. Fog Computing: Helping the Internet of Things Realize Its Potential. *Computer* **2016**, *49*, 112–116. [CrossRef]
3. Bonomi, F.; Milito, R.; Zhu, J.; Addepalli, S. Fog Computing and Its Role in the Internet of Things. In Proceedings of the first edition of the MCC Workshop on Mobile Cloud Computing, Helsinki, Finland, 17 August 2012; Association for Computing Machinery: New York, NY, USA, 2012; pp. 13–16.
4. Stojmenovic, I.; Wen, S. The Fog Computing Paradigm: Scenarios and Security Issues. In Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, 7–10 September 2014; pp. 1–8.
5. Sabireen, H.; Neelanarayanan, V.J.I.E. A Review on Fog Computing: Architecture, Fog with IoT, Algorithms and Research Challenges. *ICT Express* **2021**, *7*, 162–176. [CrossRef]
6. Kurdi, H.; Thayananthan, V. A Multi-Tier MQTT Architecture with Multiple Brokers Based on Fog Computing for Securing Industrial IoT. *Appl. Sci.* **2022**, *12*, 7173. [CrossRef]

7. *IEEE Std 1934-2018*; IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing. IEEE: Piscataway, NJ, USA, 2018; pp. 1–176. [CrossRef]
8. Qi, Q.; Tao, F. A Smart Manufacturing Service System Based on Edge Computing, Fog Computing, and Cloud Computing. *IEEE Access* **2019**, *7*, 86769–86777. [CrossRef]
9. Pfandzelter, T.; Hasenburg, J.; Bermbach, D. From Zero to Fog: Efficient Engineering of Fog-Based Internet of Things Applications. *Softw. Pract. Exp.* **2021**, *51*, 1798–1821. [CrossRef]
10. Kayal, P. Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope: Invited Paper. In Proceedings of the 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), New Orleans, LA, USA, 5–9 April 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–6.
11. Deng, R.; Lu, R.; Lai, C.; Luan, T.H.; Liang, H. Optimal Workload Allocation in Fog-Cloud Computing Toward Balanced Delay and Power Consumption. *IEEE Internet Things J.* **2016**, *3*, 1171–1181. [CrossRef]
12. Rossi, F.; Cardellini, V.; Presti, F.L. Hierarchical Scaling of Microservices in Kubernetes. In Proceedings of the 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), Washington, DC, USA, 17–21 August 2020; pp. 28–37.
13. Orive, A.; Agirre, A.; Truong, H.-L.; Sarachaga, I.; Marcos, M. Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications. *Sensors* **2022**, *22*, 1755. [CrossRef] [PubMed]
14. Nastic, S.; Rausch, T.; Scekic, O.; Dustdar, S.; Gusev, M.; Koteska, B.; Kostoska, M.; Jakimovski, B.; Ristov, S.; Prodan, R. A Serverless Real-Time Data Analytics Platform for Edge Computing. *IEEE Internet Comput.* **2017**, *21*, 64–71. [CrossRef]
15. Truyen, E.; Van Landuyt, D.; Preuveneers, D.; Lagaisse, B.; Joosen, W. A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks. *Appl. Sci.* **2019**, *9*, 931. [CrossRef]
16. Fayos-Jordan, R.; Felici-Castell, S.; Segura-Garcia, J.; Lopez-Ballester, J.; Cobos, M. Performance Comparison of Container Orchestration Platforms with Low Cost Devices in the Fog, Assisting Internet of Things Applications. *J. Netw. Comput. Appl.* **2020**, *169*, 102788. [CrossRef]
17. Red Hat OpenShift Enterprise Kubernetes Container Platform. Available online: https://www.redhat.com/en/technologies/cloud-computing/openshift (accessed on 22 May 2023).
18. TOSCA Simple Profile in YAML Version 1.3. 2020. Available online: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.pdf (accessed on 5 June 2023).
19. Wang, Y.; Lee, C.; Ren, S.; Kim, E.; Chung, S. Enabling Role-Based Orchestration for Cloud Applications. *Appl. Sci.* **2021**, *11*, 6656. [CrossRef]
20. Marchese, A.; Tomarchio, O. Communication Aware Scheduling of Microservices-Based Applications on Kubernetes Clusters. In Proceedings of the 12th International Conference on Cloud Computing and Services Science, Online, 27–29 April 2022; pp. 190–198.
21. Sebrechts, M.; Borny, S.; Wauters, T.; Volckaert, B.; De Turck, F. Service Relationship Orchestration: Lessons Learned From Running Large Scale Smart City Platforms on Kubernetes. *IEEE Access* **2021**, *9*, 133387–133401. [CrossRef]
22. Open Application Model. 2023. Available online: https://github.com/oam-dev/spec (accessed on 24 April 2023).
23. Deelman, E.; Gannon, D.; Shields, M.; Taylor, I. Workflows and E-Science: An Overview of Workflow System Features and Capabilities. *Future Gener. Comput. Syst.* **2009**, *25*, 528–540. [CrossRef]
24. Pérez, A.; Moltó, G.; Caballer, M.; Calatrava, A. A Programming Model and Middleware for High Throughput Serverless Computing Applications. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, 8–12 April 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 106–113.
25. Extend Cloud to Edge with KubeEdge | IEEE Conference Publication | IEEE Xplore. Available online: https://ieeexplore.ieee.org/document/8567693 (accessed on 15 November 2022).
26. Vayghan, L.A.; Saied, M.A.; Toeroe, M.; Khendek, F. A Kubernetes Controller for Managing the Availability of Elastic Microservice Based Stateful Applications. *J. Syst. Softw.* **2021**, *175*, 110924. [CrossRef]
27. Gkoufas, Y.; Yuan, D.Y.; Pinto, C.; Koutsovasilis, P.; Venugopal, S. Datashim and Its Applications in Bioinformatics. In *High Performance Computing, Proceedings of the ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, 2 June–2 July 2021*; Jagode, H., Anzt, H., Ltaief, H., Luszczek, P., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 416–427.
28. Wu, Y.; Wang, X. Research on Network Element Management Model Based on Cloud Native Technology. In Proceedings of the 2022 IEEE 2nd International Conference on Computer Communication and Artificial Intelligence (CCAI), Taiyuan, China, 26–28 May 2022; pp. 17–20.
29. Haja, D.; Szalay, M.; Sonkoly, B.; Pongracz, G.; Toka, L. Sharpening Kubernetes for the Edge. In Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, Beijing, China, 19–23 August 2019; pp. 136–137.
30. Ogbuachi, M.C.; Reale, A.; Suskovics, P.; Kovács, B. Context-Aware Kubernetes Scheduler for Edge-Native Applications on 5G. *J. Commun. Softw. Syst.* **2020**, *16*, 85–94. [CrossRef]
31. Zhang, X.; Li, L.; Wang, Y.; Chen, E.; Shou, L. Zeus: Improving Resource Efficiency via Workload Colocation for Massive Kubernetes Clusters. *IEEE Access* **2021**, *9*, 105192–105204. [CrossRef]

32. Katenbrink, F.; Seitz, A.; Mittermeier, L.; Müller, H.; Bruegge, B. Dynamic Scheduling for Seamless Computing. In Proceedings of the 2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2), Paris, France, 18–21 November 2018; pp. 41–48.
33. Casquero, O.; Armentia, A.; Sarachaga, I.; Pérez, F.; Orive, D.; Marcos, M. Distributed Scheduling in Kubernetes Based on MAS for Fog-in-the-Loop Applications. In Proceedings of the 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Zaragoza, Spain, 10–13 September 2019; pp. 1213–1217.
34. Workloads. Available online: https://kubernetes.io/docs/concepts/workloads/ (accessed on 10 January 2023).
35. Extending Kubernetes. Available online: https://kubernetes.io/docs/concepts/extend-kubernetes/ (accessed on 22 February 2023).
36. Sebrechts, M.; Ramlot, T.; Borny, S.; Goethals, T.; Volckaert, B.; De Turck, F. Adapting Kubernetes Controllers to the Edge: On-Demand Control Planes Using Wasm and WASI. In Proceedings of the 2022 IEEE 11th International Conference on Cloud Networking (CloudNet), Paris, France, 7–10 November 2022; pp. 195–202.
37. Operator Pattern. Available online: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (accessed on 23 January 2023).
38. KR 3 R540. Available online: https://www.kuka.com/-/media/kuka-downloads/imported/6b77eecacfe542d3b736af377562ecaa/0000270971_en.pdf (accessed on 23 October 2022).
39. ET 200SP Open Controller. Available online: https://mall.industry.siemens.com/mall/es/es/Catalog/Products/10252972 (accessed on 5 June 2023).
40. Process Simulate Software | Siemens Software. Available online: https://plm.sw.siemens.com/en-US/tecnomatix/products/process-simulate-software/ (accessed on 22 May 2023).
41. S7-PLCSIM Advanced. Available online: https://cache.industry.siemens.com/dl/files/153/109739153/att_895955/v1/s7-plcsim_advanced_function_manual_en-US_en-US.pdf (accessed on 5 June 2023).
42. K3s. Available online: https://k3s-io.github.io/ (accessed on 23 January 2023).
43. Schiraldi, M.M.; Varisco, M. Overall Equipment Effectiveness: Consistency of ISO Standard with Literature. *Comput. Ind. Eng.* **2020**, *145*, 106518. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.