Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 4, Issue 6, 2026

CONTAINER ORCHESTRATION IN FOG COMPUTING USING KUBERNETES

Sohail Anjum[*1], Khurshed Ali[2], Saif Hassan[3], Manzoor Ahmed[4], Faisal Ghaffar[5]

[*1]*Department of Electrical Engineering and Computer Science, National Chiao Tung University, Taiwan*
[2,3]*Department of Computer Science, Sukkur IBA University, Sindh, Pakistan*
School of Computer and Information Science, Hubei Engineering University, China
[5]*Department of Computer System Engineering, University of Engineering and Applied Sciences, Swat, Pakistan*

[*1]sohail.eic06g@nctu.edu.tw, [2]khurshed.ali@iba-suk.edu.pk, [3]saif.hassan@iba-suk.edu.pk, [4]manzoor.achakzai@gmail.com, [5]engr.faisal90@gmail.com

DOI: https://doi.org/10.5281/zenodo.20844849

Keywords
Cloud Computing, Fog Computing, Docker Platform, Kubernetes, Internet of Things (IoT).

Article History
Received: 26 April 2026
Accepted: 08 June 2026
Published: 21 June 2026

Copyright @Author

Corresponding Author: *
Sohail Anjum

Abstract
Over the past few years, Fog Computing Concept has emerged to improve the quality of service for the end-users to process the data closer and faster. Fog Computing is an extended version of the cloud near the user's/end devices. We propose a Fog environment for deploying micro services architecture based application, regarding micro services software architecture (SOA). Microservices based applications and software are often composed of clusters of hundreds of instances of containerized services. This cluster of containers must be fault-tolerant, available, and potentially geographically dispersed. While we are working with containers, we face some problems related to scaling up of the containers, containers' communication with each other, containers' appropriate deployment and management, auto-scaling, and distribution of traffic. Microservices is a new software development technique which is more suitable for growing IoT applications because a microservice is an independent process which fulfills the business logic. In this paper, we mainly focused on two scenarios, the first scenario is based on the development of microservice using ambient weather station and wrap up these services using the Docker container platform. Secondly, we use the orchestration platform for deployment, scaling, and management of Docker container-based microservices using Kubernetes platform. Kubernetes is a container orchestration platform. The target is to offer an efficient way to orchestrate the microservices using Kubernetes platform.

I. INTRODUCTION

Nowadays we are living in the era of technology where we have a lot of devices and sensors working together in different disciplines of daily life and we are getting more assistance with the help of these devices and make our lives smarter and easier. We are now using websites, web applications, software's and many other things related to information technology almost in every field of life like business, education, security, transportation, agriculture, airways, telecommunication sectors, which help us to solve our daily life tasks easy, faster and in a well-managed way. We can do business using the internet by single click sitting in front of laptops, we can purchase goods and pay money without going anywhere, we can secure our

homes, offices, and organization without human intervention and just using sensors. We can go for smart agriculture using different devices and grow the productivity of the land and save a lot of money and hard work. We have a large number of machines, device, and sensors which are working together in every field of life and generating huge amounts of data. To manage and handle the automation of devices, processes, and data storage, we need software applications mechanism. The data growth is very fast and it is challenging to manage and process data by using traditional Service-Oriented Architectures (Monolithic). But when we see the number of connected devices is rapidly growing (Gartner predicts about 20 billion devices by 2020 [1], Cisco even predicts about 500 billion devices to be connected to the internet by 2030 [2]), applications are expected to be changed dramatically due to the existence these massive IoT devices and embedded end-user intelligent systems by the availability of AI.

Traditionally, software's were developed for small enterprises, cooperates, and organizations as a single application with a single framework and database but application demands and new features addition make it larger, more complex and inflexible. These monolithic applications are difficult to manage, maintain, and operate due to approximately a thousand lines of code because most of the codes are tightly coupled and it is difficult to make minor changes without unintended results. This problem becomes more serious when more and more personalized applications or intelligent devices are developed. Therefore, we propose an orchestrations-based platform for deployment, scaling, and management of Docker container-based micro services.

The rest of the paper is arranged as follows: In Section II, we provide background knowledge of this work followed by related work discussion of Microservices in Section III. In Section IV, we provide a brief description of the software and tools used in our work. In Section V, we propose an Architecture of packaging up the Microservices using Container. In Section VI, we discuss the experimental results. Finally, the conclusion of this work is given in Section VII.

II. BACKGROUND

In this section, we introduce some background knowledge for this study. First, we introduce cloud computing and cloud computing-based microservices orchestration. We also provide basic knowledge and challenges in cloud-based microservices. Next, we introduce fog computing and how fog computing overcomes cloud-based issues. Next, we introduce SOA (software-oriented architectures), monolithic architecture, issues in monolithic architecture, microservices oriented architecture, the importance of microservices and challenges in microservices.

A. Cloud Computing

Cloud Computing is the delivery of computing services like servers, storage, databases, networking, software, analytics and moreover the internet. Cloud computing is based on three types of service models:

1) **IaaS – Infrastructure as a Service:** IaaS provides storage, computing, networking, and virtual resources.

2) **PaaS – Platform as a Service:** PaaS provides software development, deployment and application management.

3) **SaaS – Software as a Service:** Cloud computing application services are the most commonly utilized option for businesses in the cloud market. SaaS applications are used through a web browser directly without any installation.

Many applications require end-to-end latencies within few milliseconds, so latency is a big challenge in IoT. In this era of IoT, devices almost everywhere, we can find IoT devices generating tens of Megabytes data every few seconds. Therefore, the big problem is we need a large network capacity to transmit data. Due to low power and storage, mostly IoT devices have low computing power and limited resources, these resources are unable to fulfill the computation demands. Cloud computing has some difficulties to provide continuity, sometimes end-user devices are far away from the network at that time cloud-based authentication services will not be available [3].

B. Cloud Computing based Microservices

Nowadays cloud-based microservices is a very hot topic for researchers because of its independent architecture. Most of the applications were built-in monolithic architecture before but now the popularity of microservices architecture is higher. Cloud is good for those applications where we need to handle with massive data, more processing and we have latency issues. Cloud gives more performance as compare to Fog because we have fewer resources in Fog as compare to the cloud. However, in a sense of real-time processing and latency fog is far better than the cloud.

C. Cloud Challenges

There are a few challenges in cloud-based solutions. The first which I want to mention here is latency. Latency is a key issue between end-user and cloud communication. So for real-time applications or time-sensitive applications, we can't consider cloud because of latency. However, we can use the cloud for long term storage and for processing of massive data in which we don't have any time sensitivity problem. A real-time application such as Autonomous Cars, Smart Cities application, and Smart Healthcare, we need to minimize the latency issue but the cloud is far and latency issue is unresolvable in such scenarios. Also, Cloud is very far from end devices, the distance between them makes some connectivity problems too. To overcome latency issues, we have another layer between cloud and end devices called a fog layer.

D. Fog Computing

Fog computing is an extension of cloud computing [4]. In this layer, we have a processing unit near to the end devices for processing data with minimum latency. Fog exists between the cloud and end-users. For instance, we can say it's another layer of cloud which is next to the end-user for analyzing, processing and storing data locally. Fog computing is used for more real-time application scenarios where we have latency issues especially autonomous cars, smart cities, and smart factories.

1) How Fog Computing Overcomes Cloud Issues

Fog layer is used for real-time processing. It has a resource limitation problem but helps to resolve the latency problem. To use fog resources efficiently, we can use microservices software-oriented architecture [3]. In a microservices architecture, we can use virtualization and use the resources as per application demand. We can easily manage and assign resources from one application to another to scale up and scale down the services. For more understanding about microservices, we can go for software-oriented architecture (SOA).

E. Software Oriented Architecture

Software oriented architecture is a well-defined style to develop applications or a collection of different kinds of services. These services are communicating via different protocols and transfer the data between two or more services for performing some activity. It's very essential for one service to communicate with other services. In the SOA different services aggregate the data and perform some activity and handle the workflow for the satisfaction of user needs. SOA has many advantages such as reusability of services, if our service is independent and well-structured we can use those services in different applications. Secondly, maintenance is very simple and easy to update or modify without affecting other services. Independency to choosing a platform is also a key thing in SOA. It gives freedom to choose a platform or language. Availability, Reliability, and scalability are higher aspects of SOA.

F. Monolithic Software Oriented Architecture and its Challenges

When we talk about monolithic, it means inflexible, rigid kind of thing. But Monolithic Software architecture means a single unit of application have single data storage and performs different functions and everything is dependent on other functions or procedures and tightly coupled. When we have a single big application it creates many problems. Somehow, Monolithic architecture has its pros and cons but the cons are higher. It also depends on application size or the number of services. Monolithic architecture is

good for small applications. But for big applications, this architecture creates many problems like scaling is difficult, large code base and single data storage poses challenges in regards to modification, deployment, and maintenance as shown in Fig. 1.

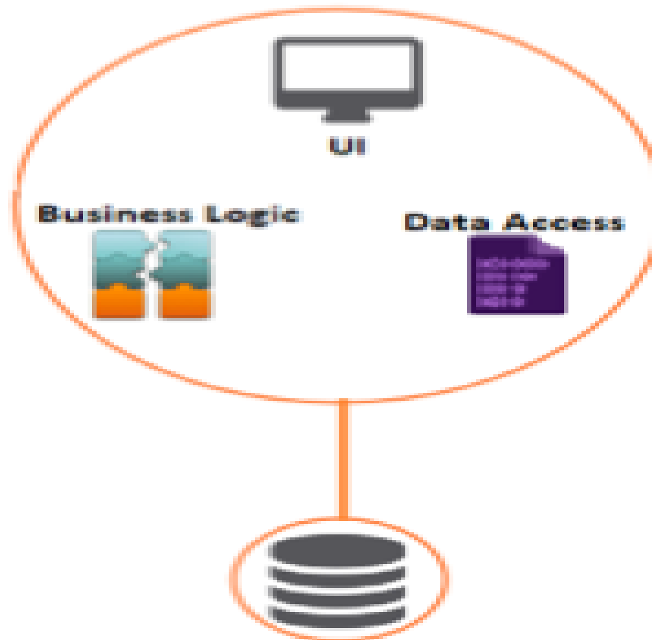

Fig. 1. A Monolithic Architecture

G. Microservices Software Oriented Architecture

Microservices is the name of a software development technique [5]. In this technique, Microservices own multiple services which perform its functionality and fulfill the business logic. Combination of services makes a microservices architecture where we have independent services connecting via API to other services. In the microservices, every service has its data storage and also it is independent and lightweight, as presented in Fig. 2.

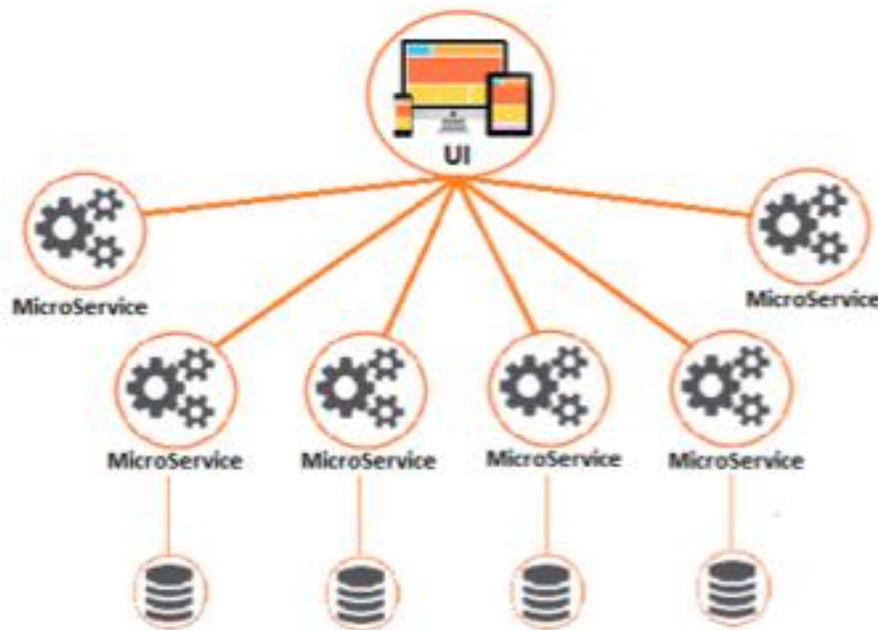

Fig. 2. A Micro Services Architecture

H. Importance of Microservices Architecture and Challenges

Importance: Microservices architecture decomposes the monolithic architecture into a set of services, but the total functionality of the application remains unchanged [6]. This solves the problem of complexity in a monolithic architecture. The application is broken into small manageable and well-defined boundaries. These services communicate with each other via API. When the application is broken into small services then there are no dependencies of services on each other, each service is developed and managed by the separate team. There is no dependency or restriction of framework or programming language, in this case, teams are free to choose the programming language and framework. Through microservices architecture, these independent services are easy to deploy. Developers don't need to wait for the other services in case if they made changes in their local service, they can deploy their local service just after testing done. Through Microservices architecture we can achieve continuous development and deployment. In Micro Services architecture application is broken, so we can easily scale up the application. If some part needs to scale up then it can be done not by changes in the other services, but in the case of monolithic architecture we need to change the whole application. Application maintenance is another great advantage of using the Micro Services Architecture.

Challenges: As it is a great saying of Fred Brooks, "there are no silver bullets". Like every architecture microservices has also some drawbacks: The name indicates the first drawback for the micro-services, it emphasizes on the size of service, as many developers consider 10–100 lines of code as a service. No doubt small services are preferred but these are not the primary function but just a mean to end. Another complexity which comes up is as Microservices architecture application is broken, we need a proper interprocess communication medium. The developer needs to write separate lines of code for handling problems in partial parts. As the lines of code are also distributed between the services, so it is quite a major challenge for handling the database in Microservices architecture application. The business transaction which has fairly a large impact on the other transactions are common, so for such business transactions, monolithic architecture is preferred because there is a single database in monolithic. Changes which span

multiple services are complex or challenging to implement in Microservices architecture. Deployment of micro serviced application is another big challenge [7]. A monolithic application is deployed on identical servers but this does not happen in case of micro serviced application.

III. RELATED WORK

Microservices software architecture modeling helps to minimize the complexities including loosely coupled and distributed services because one monolithic application could not handle the individual data generated by every single device. Microservices Software oriented architecture is easier to make changes, update, test and faster to start and stop. It's also easier to connect with other modules, even modules with different profiles in terms of processing and memory demands, easier to better source utilization and easier to construct application with different platforms and IDEs [6]. Microservices architecture is more appropriate these days for complex and large scale distributed business applications, enterprise applications as well as web applications [7]. Microservices approach is used for single application module development with small service which is lightweight, independent and communicating through HTTP resource API. Microservices are more appropriate and beneficial to handle the business capabilities due to its unique style of architecture, every single service works independently, small and not dependent on any frameworks, languages, and data storing technologies [5].

Microservice architecture has 5 key aspects mentioned as business domain knowledge and software deployment process with respect to business, automation of code which helps to build automatically and integrate, setting up the ports and define ports for every service in order to separate the logical modules, exchanging the data with each other without human intervention and last is the virtualization of each microservice container. This is a more applicable approach to build an application for supporting the scalability, fast processing, and management. In the IoT approach where we are using sensors, actuators and other physical devices, Microservices architecture is playing a key role where we need to keep the data for short time while we don't need to store that data. Microservices can be run on Fog/Edge because of minimizing the latency and bandwidth, Fog/Edge may have a processing unit for processing the data near to the end-user for data security and privacy and fast processing, but Fog/Edge have very limited processing power [8]. For easy-to-use web-based service deployment, Edge/fog devices nowadays can process complex data and detect abnormal behavior of failure of machines [9]. Microservices application is based on three main things: (1) Application Code, (2) Docker File, and (3) Application Description.

Application code is a combination of design, procedures, and functions which are developed in any programming language like Java, Python, etc. Docker file is based on the script file which includes all the dependencies and libraries which are important for running and building Docker image. The application description file is used for the application details and related information for the factory or organization which helps the user for understanding the system and its use [10]. Due to the data generation day by day we have more complex applications in the future but we can't develop new business ventures for replacing those applications to minimize the complexities. However, we can transform those monolithic applications into a microservices architecture [11]. Kubernetes is an open-source orchestration platform which facilitates us to manage the containers of Microservices and helps us to maintain, manage, schedule and scale up the Microservices [12]. Microservices help to improve the quality of the system because of containers. The container is one complete logical process which can fulfill the business need. It's lightweight and faster as compared to virtual machines. The containerized Microservices are faster to restart after failure or upgrading recovery. The deployment of Microservice-based application and orchestration using Kubernetes on a private cloud is a good approach because of its flexibility [13]. Microservices SOA provides designing and implementation guidelines for distributed applications [14]. The fog has some issues for

resource limitation so Microservices architecture is best if we want to use fog because of fewer complexities of Microservices architecture [15]. Fog was introduced by CISCO [16] to host and manage small applications. Docker platform provides containerization. The solution is that the application is segregated into small services and implemented into containers. Orchestration is a basic need for an organization [17]. Fog provides a virtualized environment where we can package up our service in a Docker container named Microservices. For orchestration, Kubernetes is the most suitable tool because of its automation and graphical user interface which is very user-friendly [18].

While the foundational works above establish the case for container-based microservices in fog/edge environments, more recent studies have refined this picture considerably. Santos et al. [19] provide a comprehensive taxonomy of fog orchestration, observing that container-based orchestration solutions — Kubernetes [19], Docker Swarm [20], Nomad, and Marathon-on-Mesos — are increasingly used to operationalize the core fog properties of heterogeneity, geographic distribution, and elastic scalability, with Kubernetes being the most frequently adopted platform in the surveyed literature. Empirical performance evaluations of four orchestration tools (Kubernetes, K3s, KubeEdge, and ioFog) specifically for edge environments show that while vanilla Kubernetes offers the richest automation feature set, its control-plane footprint is often too heavy for constrained fog nodes, motivating the use of trimmed-down distributions. This concern is echoed by surveys on container orchestration techniques tailored to real-time IoT workloads in edge and fog settings, arguing that scheduling, placement, and scaling decisions must additionally account for network latency and node heterogeneity, not just CPU/memory availability as in cloud-centric Kubernetes scheduling [21], [22].

A major trend since 2021 has been the emergence of lightweight Kubernetes distributions purpose-built for fog/edge nodes. KubeEdge extends a centralized Kubernetes control plane to large fleets of remote edge nodes and is designed to keep workloads running locally even when upstream connectivity to the cloud is lost. Comparative studies of these lightweight distributions (K3s, KubeEdge, MicroK8s, K0s) report substantial reductions in resource consumption and faster bootstrap times relative to standard kubeadm-based clusters, at the cost of some advanced scheduling and high-availability features.

Beyond raw orchestration mechanics, recent work has also moved toward intelligence-driven orchestration. Machine-learning-based container orchestration approaches, covering ML-assisted placement, predictive auto-scaling, and anomaly-aware self-healing, are identified as key future directions for fog/edge clusters with highly variable workloads. In parallel, Kubernetes-based orchestration architectures for smart-city deployments highlight cloud-edge continuum patterns where a central Kubernetes cluster coordinates many lightweight edge clusters. Security and energy-efficiency dimensions have also gained attention, with AI-enabled secure microservice orchestration at the edge and sustainability-focused studies benchmarking the power consumption of Kubernetes-based container clusters under different auto-scaling policies.

Compared to these works, the present study contributes a hands-on, fully documented deployment of a kubeadm-based Kubernetes cluster on a Xen-virtualized fog testbed, including a real IoT data source (an ambient weather station) wrapped as a Docker microservice and orchestrated alongside EdgeX Foundry microservices. While recent surveys largely focus on taxonomy, simulation, or comparative benchmarking of orchestration platforms, this work documents the practical configuration steps, common deployment pitfalls (e.g., CoreDNS CrashLoopBackOff, cgroup driver mismatches, pod network CIDR conflicts), and their resolutions — information that is often missing from higher-level survey papers but is essential for practitioners building fog-based microservice platforms.

IV. METHODOLOGY

In this section, initially, we introduce the technique of containerizing of the application using the Docker container [24], [25]. Secondly, we move towards making a cluster with different virtual machines (VMs) running on the test-bed using Xen Hypervisor. Later, we install Kubernetes [19] on it and make a cluster with master and slave node. Then, we deploy EdgeX Foundry Microservices in the cluster and show how to orchestrate the microservices using Kubernetes.

First, we introduce the scenario of development of microservice with the ambient weather station [23] and make a Docker file and containerized service using the Docker platform in Section IV-A. In Section IV-B, we explain why we consider Kubernetes platform for containerized micro-services orchestration on Fog/Edge node. Section IV-C is based on the features of Kubernetes and its usage benefits. In Section IV-D, we introduce the Kubernetes cluster architecture.

A. Application Containerization

For the experiment purpose, we developed the micro service and make a container of the application using the Docker container [24] on the fog/edge server, we use the Ambient Weather Station application [23]. Inaccuracy is one of the biggest issues while using a weather app or weather forecast station. These applications report the results to come from weather stations located at predetermined locations, typically a few miles from where we live. These applications show the changes in humidity and temperature around those stations. Therefore, we cannot get an accurate value for the temperature or humidity of your location. Solution for this problem is independent and smart weather station at your home.

Smart and independent home weather station have gained too much popularity in the past couple of years; the reason for this popularity is the extremeness of the weather conditions. In such extreme weather conditions, it is best to have its weather station. These smart and independent weather stations are not too much costly. Most of the houses are now capable of accessing such smart devices. When talking about these smart stations, Ambient WS-2902 is unbeatable. Ambient is the most affordable company for making such weather stations for home users [23]. WS-2902 ambient station is designed with such a smartness that, it provides all the features of weather conditions and gives you accurate reading using a complete kit of sensors. WS-2902 is designed so user friendly that you do not need to take the complex steps for installation this device at your home.

This station provides different types of weather condition data which is collected from different sensors. Sensors which are used in WS-2902 are temperature, humidity, UV, wind vane, anemometer, rain measurement, etc. These sensors monitor indoor and outdoor conditions, including wind speed, wind direction, rainfall, UV, solar radiation, barometric pressure, indoor/outdoor temperature (F and C), indoor humidity, dew point, heat index, wind chill and more. There is a console with this station 915 MHz RF wireless transmission, which communicate with the station. All the values from the sensors updated on the console within 16 seconds.

The main feature of this station upon which we have worked is streaming all this data to the internet. Station provides free cloud services. The console can be connected to the 2.4 GHz. Mobile application available both on iOS and Android OS for setting up the station for transmitting the real-time data to the different weather websites such as ambientweather.net, wunderground.com, and weathercloud.net. This real-time data is uploaded to these websites cloud servers and different user-friendly dashboards are available for the users. The device configuration steps are shown in Fig. 3 and Fig. 4.

Fig. 3. Ambient Device Configuration

Fig. 4. Further Steps of Ambient Device Configuration

We developed an application using Angular-JS in which is used in the microservices architecture. We use the Application Program Interface key of an ambient weather station to use it in the local server application. We use Angular JS for this because it is best for the single page application. For making any application for Docker Container [24], we need to create a Docker file in that application which is the basic pillar for showing that this application will be run as a Docker Container. Things which are mentioned in the Docker file are:

4) **Node Image:** To deploy any application as Docker Container, we need a node image for that.

5) **Running Directory:** We have to make a directory in which we are going to work on our application.

6) **Installation:** We need to install NPM and CLI in which application is developed; in our case application is developed in Angular-JS, so we use Angular CLI.

B. Kubernetes

Kubernetes [19], also called k8s, is a very famous and open-source platform developed by Google for the orchestration and management of containerized applications running in the cluster environment. Kubernetes helps for automation of application deployment, scaling of applications and management. Kubernetes works with a wide range of container tools mainly Docker containers [22], [24]. Many cloud providers offer a Kubernetes-based platform as a platform providing service.

C. Kubernetes Features

Kubernetes automatically controls the hosting of the container; Kubernetes decides where to run the containers as well as manages the multiple clusters of containers in the way of security, networking, and management [21]. Kubernetes helps to examine the health of pods and allow them to scale easily and quickly. Kubernetes mounts the volume as per your choice or needs to run the application pod; if pods have any issue Kubernetes automatically rolls back easily. Kubernetes automatically decides the place for running container to load balancing by calculating the best location. The key and beautiful feature of Kubernetes is we can run in any environment like on-premises, hybrid, and public clouds.

D. Kubernetes Cluster

The Kubernetes cluster shown in Fig. 5 has two nodes. One is master node and the other is slave node. The master nodes have GUI (graphical user interface) [26] and CLI (command-line interface), both help to control the cluster and perform the task. Kubernetes Master is based on four main components: API Server, Scheduler, Controller Manager and etcd. Slave node has four components: Service Proxy, kubelet, iptables and Container Runtime (Docker). User can access the application via service proxy. Master Node works as a controller, which can help to control all the nodes attached with the master node in the cluster. Role-based access control (RBAC) is used to manage permissions within the cluster [27].

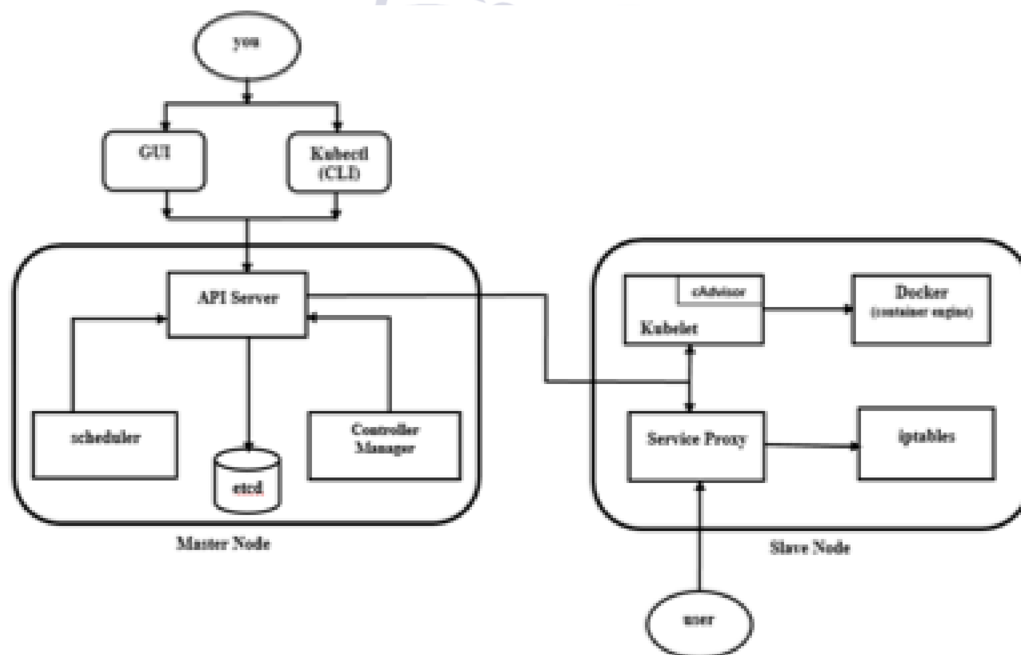

Fig. 5. Kubernetes Cluster Architecture

V. IMPLEMENTATION DETAILS

In this section, we describe the details of the application which is developed and containerized as mentioned in the previous section. Section V-A offers a hierarchical overview. Section V-B introduces the details of the scenario. Section V-C presents the implementation details.

A. Overview of Containerization of an Application

The first scenario implemented in this work concerns the development and containerization of a microservice built around the Ambient Weather Station (WS-2902) [23]. The overall workflow begins with the physical weather station collecting environmental data (temperature, humidity, UV index, wind speed and direction, rainfall, barometric pressure, and so on) through its sensor array. This data is transmitted via 915 MHz RF to the console, displayed locally, and then uploaded over the home Wi-Fi network to the Ambient Weather cloud service. An AngularJS web application was developed to consume this data through the Ambient Weather REST API and render it on a single dashboard page. To containerize this application, a Dockerfile was authored that (i) pulls a Node.js base image, (ii) creates a working directory for the application source, and (iii) installs the required NPM packages and the Angular CLI inside the image. Building this Dockerfile produces a Docker image containing the application together with all of its runtime dependencies. The resulting image was pushed to a private repository on Docker Hub [20], from which it can subsequently be pulled and instantiated as a container on any host running the Docker runtime — including the fog server used in this work. The application listens on port 4200, which is exposed when the container is run. This containerization step decouples the application from the underlying host environment, ensuring that the weather-dashboard microservice can be deployed consistently on any node of the fog cluster described in the next subsection, and forms the basic unit that is later managed by Kubernetes [19].

B. Kubernetes Orchestration Platform Architecture

While Docker [24] provides a consistent way to package and run the weather-station microservice as an isolated container, it does not by itself address multi-node scheduling, scaling, health-checking, or service discovery — all of which are required once a fog deployment grows beyond a single host. To address this, a Kubernetes cluster [19] was deployed on top of a virtualized fog testbed, structured as follows. At the base of the testbed sits a Type-1 Xen hypervisor running directly on the physical server hardware. On top of the hypervisor, two virtual machines were provisioned, each running Ubuntu Linux 16.04 LTS. These two VMs form the two logical roles of the Kubernetes cluster: one VM is designated as the Master Node and the other as the Slave (worker) Node. Both VMs were prepared identically before role-specific configuration: swap was permanently disabled (since the kubelet requires swap to be off), static IP addresses were assigned to each VM's network interface to guarantee stable cluster communication, and the Docker container runtime [22] was installed on both nodes. The Master Node hosts the Kubernetes control plane, comprising the API Server, the Scheduler, the Controller Manager, and etcd. The Slave Node hosts the components needed to actually execute workloads: the kubelet, the kube-proxy (which manages iptables rules for Pod-to-Pod and Pod-to-Service networking [21]), and the Docker container runtime itself. An overlay pod network (Weave, with CIDR range 10.244.0.0/16) was used to enable Pod-to-Pod communication across the two nodes. Finally, Docker Hub [20] was used as the image registry.

C. Implementation of Kubernetes Platform

The cluster was bootstrapped using kubeadm, following the sequence below, applied to both VMs unless otherwise noted.

- Base preparation: After updating package indexes, swap was disabled and a static IP was configured for a dedicated network interface to ensure each node has a fixed, predictable address for cluster communication.

- Container runtime installation: Docker (version 18.06.2) was installed on both nodes [22].
- Kubernetes package installation: kubeadm, kubectl, and kubelet were installed via apt-get install.
- cgroup driver configuration: Because the kubelet and the Docker runtime must agree on the same cgroup driver, the kubelet's systemd unit drop-in file was edited to explicitly set cgroup-driver=cgroupfs, matching Docker's default. Mismatches at this step were one of the most common causes of cluster initialization failure.
- Control plane initialization (Master Node only): The Master Node's control plane was initialized with kubeadm init.
- Dashboard deployment and RBAC [26], [27]: The Kubernetes Dashboard was deployed by applying its recommended manifest. A ClusterRoleBinding manifest was created and applied for administrative access.
- Joining the Slave Node: The join command was executed on the Slave VM, after which kubectl get nodes on the Master confirmed that both nodes were listed with Ready status.

With the cluster operational, the EdgeX Foundry microservices and the weather-station microservice were deployed as Kubernetes Deployment objects, each specified by a YAML manifest. These manifests were applied using kubectl apply -f <manifest-file>, after which Kubernetes scheduled the corresponding Pods onto the available nodes and exposed them according to the associated Service definitions.

VI. EXPERIMENTS

This section reports on the deployment outcomes of the two-node Kubernetes cluster described in Section V, the deployment of containerized microservices onto it, and the practical issues encountered during the experiment, along with how each was resolved.

A. Experimental Challenges

Several categories of difficulty were encountered while building the cluster. First, deploying Kubernetes [19] requires familiarity with a stack of supporting tools beyond Kubernetes itself — a hypervisor, a Linux operating system, a container runtime [22], networking plugins [21], and the kubeadm/kubectl/kubelet toolchain — and gaps in any one of these can manifest as Kubernetes-level errors that are difficult to trace back to their root cause. Second, the order of configuration steps matters. Third, version compatibility between kubeadm, kubectl, kubelet, and the Docker runtime was found to be a recurring source of subtle failures. Finally, the Kubernetes manifest format differs from Docker Compose YAML [25], despite superficial similarity, and applying Compose-style assumptions to Kubernetes manifests produced configuration errors during the EdgeX deployment.

B. Problems and Their Solutions

The following issues were the most significant encountered during deployment:

- kubeadm init failure: This was most often traced to a changed/unstable static IP address, a version mismatch between kubeadm and the installed kubelet/kubectl, or an unconfigured cgroup driver [22]. Re-checking the static IP configuration, aligning tool versions, and explicitly setting cgroup-driver=cgroupfs resolved this in all observed cases.
- CoreDNS pod in CrashLoopBackOff: CoreDNS repeatedly crashed due to a loop directive in its ConfigMap conflicting with the host's resolver configuration. Editing the coredns ConfigMap to remove the loop directive, then deleting the existing CoreDNS Pod, restored CoreDNS to a healthy Running state.
- Unable to connect to the server / TLS certificate errors: This occurred when kubectl commands were run as a non-administrative user without the KUBECONFIG environment correctly pointing to $HOME/.kube/config. Copying the admin kubeconfig to the user's home directory resolved the issue.
- kubectl proxy not starting / not reachable: This was traced to the local machine's IP address changing after the API server had already been advertised on a different address [26].
- Kubernetes APT source not found: Errors while installing kubeadm/kubectl/kubelet were traced to incorrect formatting when creating the Kubernetes list file; re-creating the file exactly as

specified by the upstream documentation resolved this.

- Pod network add-on CIDR conflicts: Running a pod network add-on (e.g., Weave) with a CIDR range that did not match the range specified during kubeadm init caused Pods to remain in Pending/ContainerCreating indefinitely [21]. Ensuring the CIDR passed to kubeadm init matched the add-on's expected range (10.244.0.0/16) resolved this.

C. Kubernetes Cluster Deployment Result

After applying the steps in Section V-C, kubectl get nodes -o wide reported two nodes — master and slave — both in Ready status, each running Ubuntu 16.04 LTS with kernel 4.15.0-54-generic and Docker 18.9.7 as the container runtime. The Kubernetes Dashboard [26], accessed via the token-authenticated proxy, displayed both nodes under the Nodes view, confirming that the control plane on the Master Node could communicate with, and schedule workloads onto, the Slave Node.

D. Kubernetes Manifest Result

A Kubernetes manifest is a declarative, YAML- (or JSON-) based description of the desired state of a resource. For the application Deployments used in this work, each manifest specifies, at minimum: the container image and tag to pull from the image registry (Docker Hub [20]), the desired number of replicas of the Pod, the container port(s) to expose, and any host-path volume mounts required by the service. As an illustrative example, the EdgeX Foundry MongoDB microservice was deployed using a Deployment manifest specifying the edgexfoundry/docker-edgex-mongo:0.2 image, one replica, container port 27017, and host-path volume mounts for /data/db, /edgex/logs, /consul/config, and /consul/data.

E. Microservice Deployment Result

Applying the manifests described above, kubectl parsed each Deployment specification and instructed the Scheduler to place the corresponding Pods on the Slave Node, which then pulled the referenced images from Docker Hub [20] and started the containers via the Docker runtime. The Kubernetes Dashboard's Overview page confirmed that all Deployments — including edgex-mongo, edgex-core-metadata, edgex-support-notifications, edgex-support-scheduler, edgex-support-logging, and the rule engine — reached 1/1 ready Pods, with corresponding ReplicaSets and Pods all reporting 100% healthy status. The previously-containerized AngularJS weather-dashboard microservice was deployed in the same manner, exposing the application on port 4200. Data originating from the Ambient Weather Station [23] was retrieved through the Ambient Weather API by the Angular application, running as a Kubernetes-managed Pod on the fog cluster's Slave Node, and rendered on the dashboard. This confirms that the full pipeline — physical sensor → cloud API → containerized microservice → Kubernetes-orchestrated deployment on a fog node — operates as intended.

VII. CONCLUSION AND DISCUSSION

This work presented an end-to-end demonstration of container orchestration for microservices in a fog computing environment using Kubernetes [19]. Two complementary contributions were made. First, a real-world IoT microservice was developed around the Ambient Weather Station WS-2902 [23], retrieving real-time environmental data through the Ambient Weather cloud API and rendering it via an AngularJS application, which was then packaged into a Docker container [24] using a custom Dockerfile. Second, a two-node Kubernetes cluster was deployed on a Xen-virtualized fog testbed using kubeadm, comprising a Master Node running the Kubernetes control plane (API server, scheduler, controller manager, etcd) and a Slave Node running the kubelet, kube-proxy, and Docker runtime, interconnected via a Weave overlay pod network. This cluster was used to orchestrate both the weather-station microservice and a set of EdgeX Foundry microservices, including a MongoDB-backed metadata service, demonstrating that Kubernetes can successfully manage Pod scheduling, networking, and health for containerized microservices on resource-constrained, geographically-local fog infrastructure rather than only on large cloud data centers.

The experimental process also surfaced a number of practical deployment challenges — cgroup driver mismatches [22], CoreDNS CrashLoopBackOff failures, TLS/kubeconfig misconfigurations, static-IP instability affecting kubectl proxy, and pod-network CIDR conflicts [21] — together with concrete resolutions for each. These findings are intended to be of direct practical value to others deploying kubeadm-based Kubernetes clusters on private, virtualized fog infrastructure, complementing the largely taxonomic or simulation-based treatment of fog orchestration found in much of the recent survey literature.

Looking forward, while the kubeadm-based deployment demonstrated here is functionally complete, its control-plane footprint is non-trivial for genuinely resource-constrained fog/edge hardware. Lightweight Kubernetes distributions such as K3s and KubeEdge have emerged specifically to address this gap, offering substantially reduced memory and CPU overhead while retaining API compatibility with standard Kubernetes manifests; migrating the cluster presented here to such a distribution would be a natural next step for deployment on true edge hardware (e.g., single-board computers) rather than virtual machines. Additionally, as fog deployments scale beyond two nodes and begin to host dozens of heterogeneous microservices with time-varying load — as is expected in domains such as smart healthcare, smart cities, and autonomous vehicles — static, resource-based scheduling becomes insufficient. Incorporating machine-learning-based placement and auto-scaling policies represents a promising direction for ensuring that fog-based Kubernetes clusters can meet the latency and availability requirements of next-generation, real-time IoT applications. We hope that the implementation details, deployment workflow, and troubleshooting guidance documented in this work provide a practical foundation for future fog-based microservice orchestration systems.

REFERENCES

[1] Gartner, “IoT eBook: Internet of Things,” https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf, 2014, accessed: 2024.

[2] Cisco Systems, “Internet of Things At-a-Glance,” https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45731471.pdf, 2020, accessed: 2024.

[3] M. Chiang and T. Zhang, “Fog and IoT: An overview of research opportunities,” IEEE Internet of Things Journal, vol. 3, no. 6, pp. 854–864, 2016.

[4] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, “A comprehensive survey on fog computing: State-of-the-art and research challenges,” IEEE Communications Surveys & Tutorials, vol. 20, no. 1, pp. 416–464, 2017.

[5] D. Namiot and M. Sneps-Sneppe, “On micro-services architecture,” International Journal of Open Information Technologies, vol. 2, no. 9, pp. 24–27, 2014.

[6] D. S. Linthicum, “Practical use of microservices in moving workloads to the cloud,” IEEE Cloud Computing, vol. 3, no. 5, pp. 6–9, 2016.

[7] Y. Xie, X. Zhou, H. Xie, G. Li, and Y. Tao, “Research on the architecture and key technologies of integrated platform based on micro service,” in 2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC). IEEE, October 2018, pp. 887–893.

[8] B. Butzin, F. Golatowski, and D. Timmermann, “Microservices approach for the internet of things,” in 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, September 2016, pp. 1–6.

[9] J. Tuvakov and K. Park, “On the fog node model for multi-purpose fog computing systems,” in 2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). IEEE, November 2018, pp. 1211–1214.

[10] J. Ha, J. Kim, H. Park, J. Lee, H. Jo, H. Kim, and J. Jang, "A web-based service deployment method to edge devices in smart factory exploiting Docker," in 2017 International Conference on Information and Communication Technology Convergence (ICTC). IEEE, October 2017, pp. 708–710.

[11] S. Sebastian, "Transform monolith into microservices using Docker," in 2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA). IEEE, August 2017, pp. 1–5.

[12] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with Kubernetes: Experiments and lessons learned," in 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, July 2018, pp. 970–973.

[13] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures – a technology review," in 2015 3rd International Conference on Future Internet of Things and Cloud. IEEE, August 2015, pp. 379–386.

[14] N. Alshuqayran, N. Ali, and R. Evans, "Towards micro service architecture recovery: An empirical study," in 2018 IEEE International Conference on Software Architecture (ICSA). IEEE, April 2018, pp. 47–4709.

[15] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, "Fog computing: Survey of trends, architectures, requirements, and research directions," IEEE Access, vol. 6, pp. 47 980–48 009, 2018.

[16] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16.

[17] J. Rufino, M. Alam, J. Ferreira, A. Rehman, and K. F. Tsang, "Orchestration of containerized microservices for IIoT using Docker," in 2017 IEEE International Conference on Industrial Technology (ICIT). IEEE, March 2017, pp. 1532–1536.

[18] M. S. de Brito, S. Hoque, T. Magedanz, R. Steinke, A. Willner, D. Nehls, and F. Schreiner, "A service orchestration architecture for fog-enabled infrastructures," in 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC). IEEE, May 2017, pp. 127–132.

[19] The Kubernetes Authors, "Kubernetes," https://kubernetes.io/, accessed: 2026.

[20] Docker Inc., "Docker Hub," https://hub.docker.com/, accessed: 2026.

[21] The Kubernetes Authors, "Network plugins," https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/, accessed: 2026.

[22] The Kubernetes Authors, "Container runtimes," https://kubernetes.io/docs/setup/production-environment/container-runtimes/, accessed: 2026.

[23] Ambient Weather, "Ambient Weather – Professional Weather Stations, Wireless Weather Sensors," https://www.ambientweather.com/, accessed: 2026.

[24] Docker Inc., "What is a container?" https://www.docker.com/resources/what-container/, accessed: 2026.

[25] Docker Docs, "Docker Compose Overview," https://docs.docker.com/compose/, accessed: 2026.

[26] The Kubernetes Authors, "Web UI (dashboard)," https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/, accessed: 2026.

[27] The Kubernetes Authors, "Using RBAC authorization," https://kubernetes.io/docs/reference/access-authn-authz/rbac/, accessed: 2026.