Secondary Publication

Wen, Long; Zhang, Yu; Rickert, Markus; u. a.

Cloud-Native Fog Robotics : Model-Based Deployment and Evaluation of Real-Time Applications

Date of secondary publication: 29.08.2025

Version of Record (Published Version), Article

Persistent identifier: urn:nbn:de:bvb:473-irb-109944x

Primary publication

Wen, Long; Zhang, Yu; Rickert, Markus; u. a. (2025): Cloud-Native Fog Robotics : Model-Based Deployment and Evaluation of Real-Time Applications, in: IEEE Robotics and automation letters, New York, N.Y.: Institute of Electrical and Electronics Engineers (IEEE), Vol. 10, Nr. 1, pp. 398–405, doi: 10.1109/LRA.2024.3504243.

Legal Notice

This work is protected by copyright and/or the indication of a licence. You are free to use this work in any way permitted by the copyright and/or the licence that applies to your usage. For other uses, you must obtain permission from the rights-holders.

This document is made available under a Creative Commons license.

The license information is available online:

https://creativecommons.org/licenses/by/4.0/legalcode

Cloud-Native Fog Robotics: Model-Based Deployment and Evaluation of Real-Time Applications

Long Wen, *Student Member, IEEE*, Yu Zhang, Markus Rickert, *Member, IEEE*, Jianjie Lin, Fengjunjie Pan, and Alois Knoll, *Fellow, IEEE*

***Abstract*—As the field of robotics evolves, robots become increasingly multi-functional and complex. Currently, there is a need for solutions that enhance flexibility and computational power without compromising real-time performance. The emergence of fog computing and cloud-native approaches addresses these challenges. In this paper, we integrate a microservice-based architecture with cloud-native fog robotics to investigate its performance in managing complex robotic systems and handling real-time tasks. Additionally, we apply model-based systems engineering (MBSE) to achieve automatic configuration of the architecture and to manage resource allocation efficiently. To demonstrate the feasibility and evaluate the performance of this architecture, we conduct comprehensive evaluations using both bare-metal and cloud setups, focusing particularly on real-time and machine-learning-based tasks. The experimental results indicate that a microservice-based cloud-native fog architecture offers a more stable computational environment compared to a bare-metal one, achieving over 20% reduction in the standard deviation for complex algorithms across both CPU and GPU. It delivers improved startup times, along with a 17% (wireless) and 23% (wired) faster average message transport time. Nonetheless, it exhibits a 37% slower execution time for simple CPU tasks and 3% for simple GPU tasks, though this impact is negligible in cloud-native environments where such tasks are typically deployed on bare-metal systems.**

***Index Terms*—Hardware-software integration in robotics, software architecture for robotic and automation.**

I. Introduction

THE field of robotics is advancing rapidly, requiring increased computational power and storage for complex, data-intensive tasks. Cloud robotics addresses this need by leveraging the vast computing resources of cloud platforms, enabling robots to access extensive data processing capabilities and advanced algorithms. However, limitations in network infrastructure and geographical factors can introduce latency, undermining the stability and efficiency of cloud-based systems. Fog robotics presents a strategic solution by applying fog computing principles: decentralizing data processing and bringing it closer to robots in order to reduce latency, which is crucial for real-time applications. Compared with similar edge computing, which relies on individual devices with limited computational resources, fog computing leverages fog servers with higher processing power, allowing for more complex data analytics without overburdening edge devices. While edge computing faces scalability challenges as it depends on each device to process data, fog computing introduces a middle layer that offers centralized coordination, improving system management. This makes fog computing particularly suitable for complex robotic applications that require real-time performance.

Recent advancements in robotics have enabled robots to incorporate capabilities such as control, perception, simulation, and planning. As robotic applications scale, system complexity increases, presenting challenges in deployment and maintenance [1]. Monolithic architectures exacerbate these challenges by reducing system adaptability, making updates and modifications difficult. Moreover, as systems expand, troubleshooting and maintenance become more complex, often leading to inefficient resource utilization. When complex robotic systems integrate with cloud-native fog architecture, these challenges are further amplified. Given this, there is a pressing need to shift towards a microservice architecture, a predominant architectural style in service-oriented software [2], [3], exemplified by the Robot Operating System (ROS). Transitioning to a microservice-based architecture fragments the monolithic structure into smaller, more manageable services, increasing the number of components. This increase, however, complicates resource allocation and software deployment. Efficient orchestration and containerization strategies are thus crucial for managing and deploying these services effectively.

In this paper, we propose an enhanced microservice architecture for fog robotics by decomposing ROS-based complex robotic systems into smaller, manageable components and leveraging containerization to create isolated execution environments for each component, as illustrated in Fig. 1. We utilize Kubernetes, specifically its k3s variant optimized for embedded systems, as the central tool for orchestrating these

Received 15 July 2024; accepted 9 November 2024. Date of publication 21 November 2024; date of current version 5 December 2024. This article was recommended for publication by Associate Editor A. Dutta and Editor M. A. Hsieh upon evaluation of the reviewers' comments. *(Corresponding author: Yu Zhang)*

Long Wen, Yu Zhang, Fengjunjie Pan, and Alois Knoll are with the Robotics, Artificial Intelligence and Real-Time Systems, School of Computation, Information and Technology, Technical University of Munich, 80333 Munich, Germany (e-mail: wenl@in.tum.de; zha1@in.tum.de, zy.zhang@tum.de; panf@in.tum.de; knoll@in.tum.de).

Markus Rickert is with the Multimodal Intelligent Interaction, Faculty of Information Systems and Applied Computer Sciences, University of Bamberg, 96047 Bamberg, Germany (e-mail: markus.rickert@uni-bamberg.de).

Jianjie Lin is with the Mercedes-Benz Group, RD/ASF Driver Abstraction, 70372 Stuttgart, Germany (e-mail: jianjie.lin@mercedes-benz.com).

Video is available here: https://www.youtube.com/watch?v=gN5tykUIFfw.

This letter has supplementary downloadable material available at https://doi.org/10.1109/LRA.2024.3504243, provided by the authors.

Digital Object Identifier 10.1109/LRA.2024.3504243

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

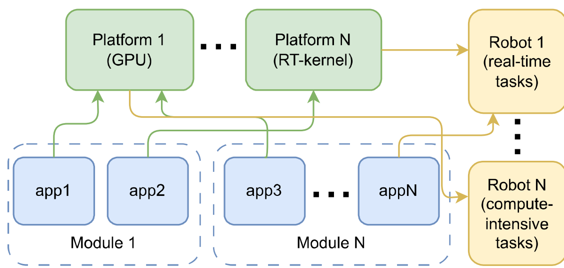


Fig. 1. To integrate enhanced microservices into fog robotics, monolithic modules are segmented into independent apps, deployed in dedicated containers, and assigned to platforms with different configurations.

containers in the distributed fog robotics system, thereby crafting a microservice-based cloud-native fog robotic architecture. Furthermore, we adopt a modular approach grounded in model-based systems engineering (MBSE) to streamline the configuration and modification of the architecture. By employing MBSE, we align with the design principles of microservice-based architectures, substantially enhancing the structure's reusability, clarity, maintainability, traceability, and operational efficiency. These strategies enable the independent deployment, restarting, or updating of each component, simplifying the process of managing and deploying the system while easily integrating new functionalities. The feasibility and performance of these strategies are demonstrated by comprehensive experiments in real-time scenarios.

The contributions of this work are summarized as follows:

- We propose a solution for managing complex fog robotic systems by integrating the microservices modular approach with the cloud-native fog robotics systems to enhance their flexibility and manageability.
- A MBSE-based modeling strategy is provided to streamline the architecture's configuration, modification, and resource allocation.
- Our approach is evaluated in real-time application scenarios, demonstrating a more stable computational environment with 20% reduction in standard deviation for complex algorithms, faster startup times, 17% (wireless), and 23% (wired) faster message transport.

The remainder of this paper is organized as follows. Section II reviews the research background in fog robotics and microservice architectures. Section III details the proposed microservice architecture and the role of MBSE and containerization. Section IV demonstrates the feasibility of our approach and evaluates it under real-time scenarios. Section V discusses the advantages and limitations of our approach.

II. Related Work

Offloading the heavy computational load to cloud servers enables robotic systems to perform more advanced tasks [4]. Typical examples are RoboEarth [5] and Robot-cloud [6], which address the computational limitations of low-cost robots. However, factors such as latency, availability, and security limit the usage of cloud robotics.Tian et al. [7] proposed a fog robotic system by combining cloud robotics with an agile edge device to overcome these issues. Giovanni et al. [7] proposed DewROS2, a platform for fog Robotics that enables real-time system monitoring with minimal performance impact. Tanwani et al. [8] also employed fog robotics in their system for deep robot learning, effectively reducing inference time. The FogROS2 proposed by Ichnowski [9] integrates cloud and fog robotics within the ROS distribution, enhancing performance and timing compared to ROS2.

These robotic systems are complex and will likely become more so as robotics technology advances. To mitigate this complexity and simplify management, we employ the concept of microservices. Microservices address some of the limitations of Service Oriented Architecture (SOA) with a more refined and modular approach [10]. Luis et al. explored the feasibility of microservices in service orchestration using cloud-native technology, highlighting its benefits and challenges in real-world scenarios [11]. Similarly, Singh investigated the use of microservice architecture in cloud applications, evaluating its performance in terms of response and deployment times [12]. In our previous work [13], we observed that microservice could deliver improved startup performance in the field of autonomous driving. In the field of robotics, Xia et al. [14] presented a microservice-based service management architecture for cloud robotic applications. However, these works focus either solely on cloud-native technology or microservice-based architecture. Currently, there is a lack of research on integrating fog robotics with microservices and containerization, enabling efficient modification and management of robotic systems.

Several studies have provided solutions for resource allocation challenges using MBSE. Pohlmann et al. [15] introduced MechatronicUML, a methodology tailored for managing resource allocation in distributed vehicular systems. Al-Azzoni et al. [16] developed a flexible framework that allows users to define custom models for describing their systems, aiming to address resource allocation issues effectively. Nevertheless, there is no practical solution grounded in established systems and software engineering standards that can be seamlessly applied to a microservice-based fog robotic system while also providing the flexibility to define specific requirements.

III. Methodology

This section illustrates the steps for establishing a microservice-based cloud-native fog architecture from scratch, its modeling process, and how we decompose a monolithic robotic system.

A. Overview of the MBSE Method

By integrating MBSE with containerization and microservices, as illustrated in Fig. 2, we aim to simplify and automate system creation and deployment, while enhancing flexibility and scalability. Our method begins by collecting hardware and software information of the robotic system, which is used to create the meta-model–a high-level abstraction that defines the essential rules, meta-types, and properties required for semantic model creation. Subsequently, we instantiate this meta-model by creating a partial instance model, effectively decomposing monolithic applications into microservices that specify the required hardware and software environments. Based on these two models, we establish design constraints according to the requirements and convert them into Satisfiability Modulo Theories (SMT) problems. Solving these SMT problems enables the generation of a solution instance model with explicit resource allocations, thereby providing a concrete representation

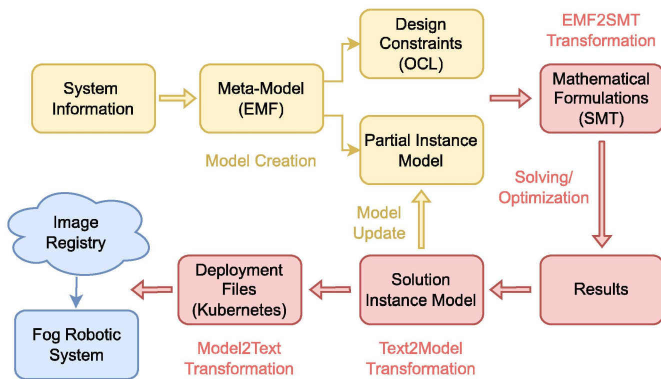


Fig. 2. In the end-to-end workflow for the MBSE method, the yellow parts are modeling steps that require manual effort, while the red ones should be automated after model creation.

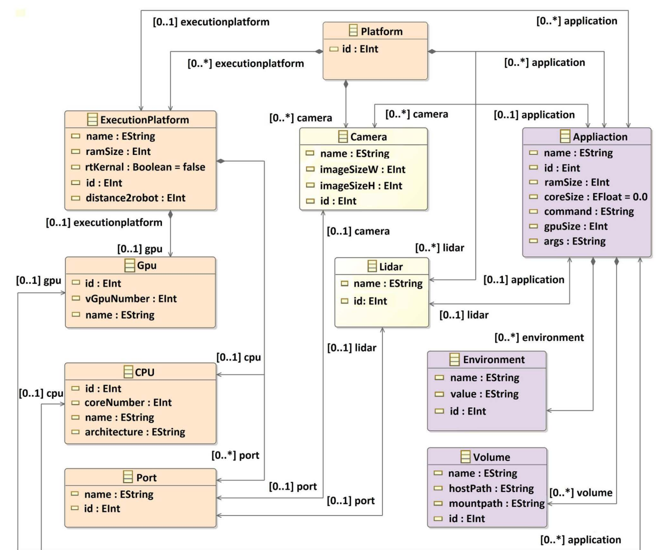


Fig. 3. A meta-model designed for the cloud-native fog robotics setup, hardware, software, and external devices are highlighted in orange, purple, and yellow, respectively.

of the target system. When new components, such as sensors or robots, need to be integrated into the system, the software and hardware requirements of the decomposed applications for these components are added to the solution instance model. The design constraints are then redefined, and the SMT problems are resolved to produce an updated solution instance model. Finally, the solution instance model is transformed into deployment files, facilitating the deployment of the fog robotic system using Kubernetes. This architecture allows each component to be independently modified, restarted, and redeployed, thereby meeting specific system requirements with enhanced flexibility and scalability.

B. Modeling Method

The Eclipse Modeling Framework (EMF) is selected as our modeling tool, providing a structured methodology for documenting and constructing the system. The initial step in establishing the meta-model involves abstracting the hardware components of the robotic system. To achieve this, we define an *ExecutionPlatform* class, incorporating properties such as name, RAM size, and real-time kernel availability, to represent various PCs and embedded systems, as illustrated in Fig. 3. Subsequently, we create *Gpu*, *Cpu*, and *Port* classes to provide detailed hardware information for each *ExecutionPlatform*.

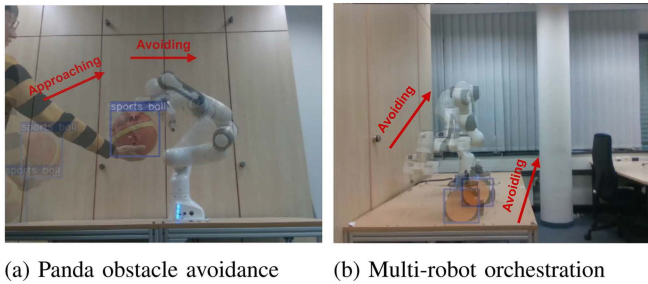
The software information is abstracted through defining the *Application* class with properties like RAM, CPU, and GPU requirements, as well as the commands to be executed upon creation. This class describes the different applications that make up the cloud-native robotic system. Additionally, we define the *Environment* and *Volume* classes to specify the environmental arguments and volume mounts required by the applications. Apart from the execution platform and applications running on it, there are external devices, such as sensors, in a robotic system. In this paper, *Camera* and *Lidar* classes are created to represent the sensors utilized in the system. Finally, we establish associations between classes to illustrate relationships. For instance, one application can have only one execution platform, while one platform can host multiple applications.

Upon completing the meta-model, we employ an instance model to instantiate it. During the instance modeling phase, we decompose the robot's control components and associated algorithms into smaller functional segments, thereby aligning with a microservice-based architecture. The instance model provides detailed definitions for each decomposed application, encompassing both software and hardware requirements. Specifically, each application specifies the required container image, the command to execute upon container creation, environment variables, and necessary volume mounts. This methodology enables a clear and precise description of each component within the complex robotic system, resulting in a set of more manageable applications.

Modifying the instance model facilitates the adjustment of resource allocation and the addition or removal of applications, sensors, and PCs easily. However, manually configuring resource allocation for each application in a complex robotic system is impractical. Therefore, design space exploration is needed by establishing the design constraints. For this purpose, we employ the Object Constraint Language (OCL), a declarative formal language developed by the Object Management Group (OMG). Fig. 4 illustrates the constraints utilized in this work. In addition to the constraints depicted in the figure, our design constraints include specific requirements such as assigning one CPU per application, allocating each application to the appropriate platform, and assigning GPU resources as needed.

```
#Keep resource assignments within limits.
context Cpu
  inv cpuCore:
  application -> collect(coreSize) -> sum() <=
      coreNumber
context Gpu
  inv gpuPower:
  application -> collect(gpuSize) -> sum() <=
      vGpuNumber
#Ensure apps have access to required devices.
context Application
  inv appLidar:
  lidar -> forAll(port.executionplatform = self.
      executionplatform)
```

Fig. 4. OCL Constraints for the instance model.



(a) Panda obstacle avoidance (b) Multi-robot orchestration

Fig. 5. Real world setup of the experiments.

After defining the constraints, we leverage the EMF2SMT transformation method, as introduced in our previous work [17]. This technique facilitates the conversion of EMF models and OCL specifications into SMT problems. We utilize the SMT-Lib format to present these problems. Here, we convert the SMT problems generated using OCL constraints, as shown in Fig. 4, into human-readable formulas:

$$\sum_{a \in \mathcal{A}} g_a \leq \mathcal{G} \tag{1}$$

where $\mathcal{A}$ and a denote the set of applications and a specific application, g_a is the amount of GPU resource required by an application, and $\mathcal{G}$ is the total available GPU resources.

$$\sum_{a \in \mathcal{A}} c_a \leq \mathcal{C} \tag{2}$$

where c_a is the CPU resources required by an application, $\mathcal{C}$ is the total available CPU resources. The last formula is:

$$x_{a \to d} \Rightarrow \varphi ::= \mathbf{true} \tag{3}$$

where $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$, φ denotes whether a is suitable on a platform, $\varphi_1 = x_{a \to d}, \forall a \in \mathcal{A}, d \in \mathcal{D}$, φ_1 is a binary variable indicating whether a requires d, $\mathcal{D}$ and d denote the device set and a specific device, respectively, $\varphi_2 = x_{pd \to e}, \varphi_3 = x_{a \to e}, e \in \mathcal{E}$, φ_2 and φ_3 indicate whether a port or a is mapped to e, $\mathcal{E}$ and e is the set of execution platform and one specific platform. These SMT problems can then be processed by state-of-the-art solvers like *Z3*. We plug these results into the instance model employing Eclipse Acceleo. Then, the Meta-Object Facility Model to Text (MOFM2T) transformation will be utilized to generate Kubernetes deployment files. Through this approach, we efficiently and reliably obtain the distribution of different applications among various PCs within complex fog robotic systems while adhering to specified requirements.

C. Combining Microservice Architecture With MBSE Method

For better illustration, we employ real-world experimental setups, as illustrated in Fig. 6, to construct a microservice-based fog robotic system using the MBSE approach. In the first experiment involving a single Franka robot, we develop both a meta-model and an instance model. The meta-model construction follows the methodology outlined in Section III-B. When defining the instance model, we first decompose the monolithic robot and sensor modules into microservices by function. Specifically, the Franka real-time interface module is decomposed into five primary applications: *Gripper* for controlling the gripper mechanism, *Publisher* for broadcasting the robot's state, *Manager* for managing controller switching, *TF* for transformation tree handling, and *Controller* for operating various controllers. The camera application is responsible for environment perception, while the detection application handles the data it collects. The obstacle avoidance algorithm is split into two applications, one requiring GPU access and the other only CPU. Subsequently, each application was specified with its hardware and software requirements, including CPU, GPU, ports, sensors, environment variables, and volume mounts, within the partial instance model. Following the process in Section III-B, we generated the final deployment file to deploy these nine applications into their dedicated containers.

In the second experiment, we integrated both a Turtlebot and a Franka robot into the workspace. To incorporate the Turtlebot, we first defined a new *Lidar* class in the meta-model to abstract the additional sensor introduced by the Turtlebot. We then decomposed the Turtlebot control into two applications: *Turtlebot Core* (responsible for rosserial connections to Lidar, motor, and power drivers) and *Diagnose* (for error monitoring). We also introduce the clustering application for processing the lidar data and obstacle avoidance application to prevent collision. These new applications–*Turtlebot Core*, *Diagnose*, *clustering*, *obstacle avoidance* and *Lidar*–were added to the instance model. The remaining process, as outlined in Section III-B, was repeated to generate the updated deployment file for the extended robotic system with 15 different applications. These two configurations of robotic systems demonstrate that incorporating a new robot or sensor into the system requires only a few modifications to the instance and meta-model, highlighting the efficiency and flexibility of our approach.

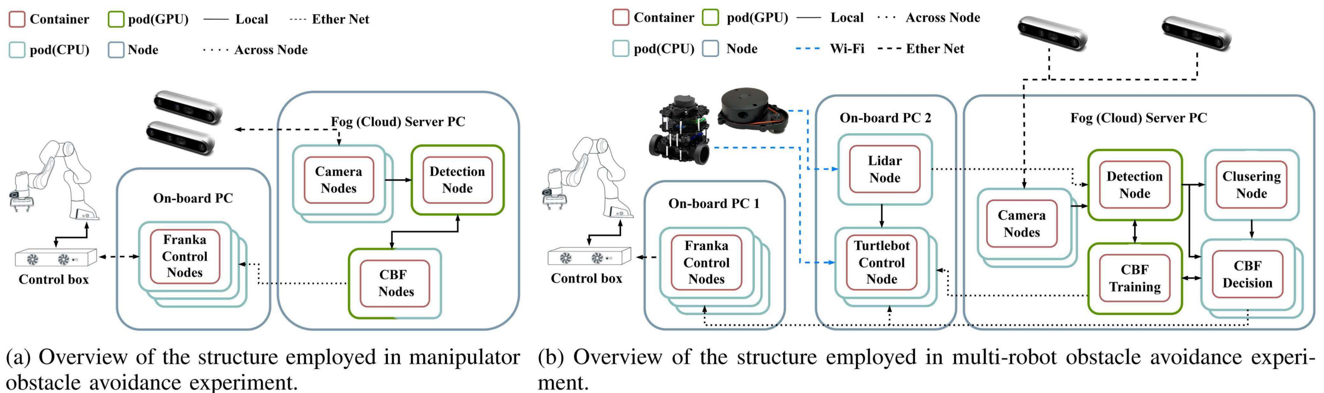
IV. Experiments

This section examines the impact of the microservice-based fog robotic architecture in practical scenarios.

A. Experiment Setup

Our experiments employ the Control Barrier Functions (CBFs) methodology [18] to deploy an obstacle avoidance algorithm among robots. These setups demand real-time control with a high frequency of at least 100 Hz, presenting a significant challenge to the containerized environment inherent in a microservice-based cloud-native fog (cloud) architecture. A comprehensive description of the CBF-based algorithm utilized in this paper is detailed in [19].

The structure of our two experimental setups is illustrated in Fig. 6(a) and (b). The first setup involves a single Panda robot, while the second setup includes a Franka robot and a Turtlebot performing obstacle avoidance simultaneously in the same workspace. Each robot is equipped with an onboard PC, complemented by two cameras and a Lidar for environment perception. We have integrated microservices into the cloud-native fog architecture, as previously described. We carefully divided the modules into distinct applications based on their functions (as detailed in Section III) to optimize these setups. Our evaluation compares the cloud-native fog architecture with the traditional monolithic setup (hereafter referred to as *bare-metal*) and the cloud setup. The cloud setup is almost the same as the fog one, but we utilize *tc qdisc* package to simulate the latency introduced by cloud setup. The latency is set to 30 ms while the jitter is 20 ms.



(a) Overview of the structure employed in manipulator obstacle avoidance experiment.

(b) Overview of the structure employed in multi-robot obstacle avoidance experiment.

Fig. 6. Overview of the structures employed in experiments. The Franka Control is divided into 5 components: Gripper, Manager, Publisher, TF and Controller; the Turtlebot Control is divided into 2 components: Vehicle and Diagnostic.

It is important to note that in our experiment, the bare-metal setup includes not only the onboard PC but also the server PC with the help of ROS. This means the bare-metal configuration is also a fog setup but without separating into microservices and containerization. Using only the onboard PC for the bare-metal setup would result in the fog/cloud setups significantly outperforming it, which would not be a fair comparison. Our goal is to investigate the impact of containerization and the division of the monolithic setup. Therefore, we chose a conventional fog setup as the 'bare-metal' reference.

The experimental setup includes a PC equipped with an Intel i5-6500 CPU and a Ubuntu real-time kernel serving as an onboard computer of Franka. There is a more powerful workstation featuring an Intel i9-12900 K CPU and an RTX 3080 Nvidia GPU that acts as the server. The controlled robots are a Franka Panda robot and a Turtlebot3 burger vehicle with a Raspberry Pi 4B as an onboard PC. The real-world setup for the experiments is shown in Fig. 5. All experiments are repeated 20 times to ensure a reliable result, and the results presented are the average among all runs.

B. Evaluation Metrics

The primary benefits of a microservice-based cloud-native fog architecture include flexible deployment and enhanced failure recovery capabilities. Accordingly, we have chosen startup time as a critical metric for evaluating performance. This includes the overall system startup time and the startup time for individual functions. We define the overall startup time as the interval between command input and algorithm execution. Function startup time is measured from the initialization of the first node in a function to the completion of the initialization of the last node in that function. Additionally, we assess the latency in message transmission, the execution time of algorithms, and overall resource consumption to provide a comprehensive evaluation of the performance.

C. Franka Robot Manipulator Obstacle Avoidance

This experiment mainly focuses on the robot manipulator. It investigates the performance of cloud-native fog robotic architecture (hereafter referred to as *fog setup*) in real-time (1000 Hz for robot manipulator, 100 Hz for CBF-based algorithm) obstacle avoidance tasks.

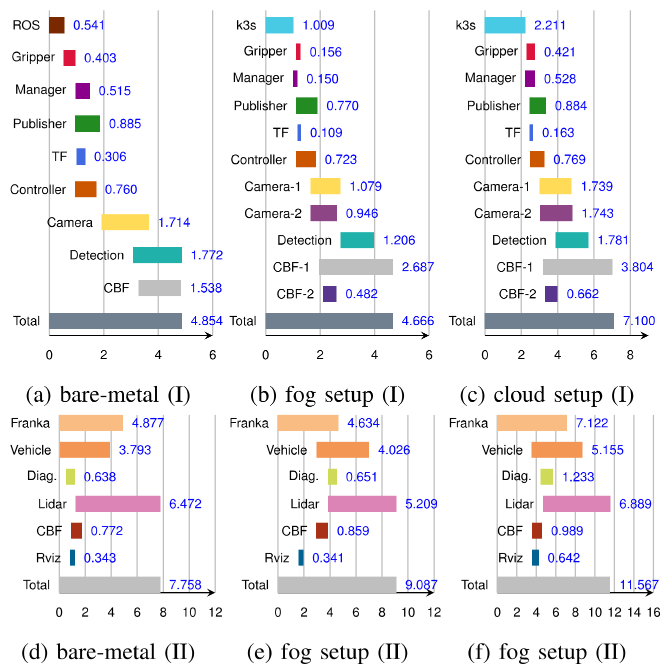
TABLE I
THE TRAINING TIME (IN MS) OF GP UNDER FOG/CLOUD SETUP AND BARE-METAL

Experiment	Setup	Mean	Min	Max	Std.
OnT CPU	bare-metal	0.294	0.069	12.761	0.378
	fog	0.467	0.098	12.733	0.663
	cloud	32.381	20.908	52.773	6.682
OnT GPU	bare-metal	0.692	0.117	14.137	0.803
	fog	0.715	0.116	17.260	0.907
	cloud	33.099	20.100	58.072	6.979
OffT CPU	bare-metal	3212.6	3128.2	3350.8	109.8
	fog	3192.9	3133.8	3337.8	79.8
	cloud	3236.7	3158.6	3381.2	114.5
OffT GPU	bare-metal	2376.6	2278.1	2434.1	60.5
	fog	2379.7	2332.9	2485.2	45.2
	cloud	2401.3	2357.2	2542.1	66.7

OnT represent *online training, OffT* represent *offline training, Std.* Represent *standard deviation*.

1) Training Time: The CBF-based algorithm initially requires offline training to optimize the hyperparameters of the kernel function used in Gaussian Regression. Once this setup is complete and the hyperparameters are fixed, it enters a phase where it dynamically updates other real-time parameters based on live environmental data. The following text will refer to this phase as *online training*. Table I presents the results for the CBF-based algorithm across different setups, and the best metrics are highlighted in green.

The initial observation reveals that the online training outcomes from the cloud setup are inadequate for real-time tasks. The training time significantly exceeds those observed in both fog and bare-metal setups. Such latency hinders proper and agile control of the robot, highlighting the incompatibility of cloud architecture with tasks demanding real-time performance. For online training, the bare-metal architecture significantly outperforms (37%) the fog setup when utilizing the CPU. However, in cloud-native environments, tasks requiring less than one millisecond are typically deployed on bare-metal systems. As a result, the reduced performance for these tasks does not impact the overall performance of the proposed architecture. The performance gap between fog and bare-metal architectures narrows when GPU is used for online training with only 3% lower mean execution time. CPU and GPU environments exhibit similar performance for offline training across the three architectures. Notably, the fog architecture demonstrates a 25-28%



(a) bare-metal (I) (b) fog setup (I) (c) cloud setup (I)

(d) bare-metal (II) (e) fog setup (II) (f) fog setup (II)

Fig. 7. Comparison of Startup time (in seconds) under bare-metal, fog/cloud setup in two experiments(I, II). *Franka* stands for the startup time of Franka robot.

lower standard deviation in offline training times, indicating more stable performance than bare-metal.

2) Startup Time: Startup times between the fog setup and bare-metal are compared in Fig. 7(a)–(c). The initial observation indicates that the startup time for the fog architecture is 0.2 s faster than that of the bare-metal setup, while the cloud setup is 2.2 s slower. Additionally, in the bare-metal setup, the applications are launched one after another, whereas in the fog/cloud setup, they are launched almost simultaneously. When the Franka real-time interface is segmented into five smaller components, each component launches more rapidly than when integrated as a single unit, particularly the Gripper, Manager, and TF applications. Even the cloud setup can deliver comparable performance. This does not mean the fog/cloud setup performs significantly better than the bare-metal setup. We further elaborate on this by using the TF application as an example. The TF application contains only one node that publishes the transformation information between joints and the gripper. Launching only this node in a fog setup inside a dedicated container takes little time. Conversely, in a bare-metal setup, ROS is tasked with launching a suite of applications at one time. It must handle prerequisite steps such as setting up the parameter server, launching all associated dependent nodes, and more before it can proceed with the task after launching the TF node. Therefore, the TF's launch time in a bare-metal environment includes the time taken by the prerequisite steps. This is the reason why these applications have extended launch times on bare-metal. However, the fog setup does deliver a more efficient startup time. Specifically, the Franka real-time interface in the fog environment launches in just 0.9 s, outpacing the bare-metal's 1.3 s. The CBF-based algorithm exhibits a 1.1 s longer launch time under the fog architecture. Despite this, the earlier launch of the CBF-based algorithm in the fog setup compensates for the delay, resulting in a faster overall startup time. In conclusion, the fog architecture demonstrates a more efficient startup process in the experiments conducted. Additionally, the fog setup is more convenient, requiring only a single terminal command, whereas the bare-metal setup requires extra manual operations.

TABLE II
THE IMAGE TRANSMISSION TIME (IN MILLISECONDS) BETWEEN SENSOR NODES AND DETECTION NODE

Experiment	Setup	Mean	Min	Max	Std.
Camera 1	bare-metal	2.505	0.251	14.079	3.782
	fog	1.939	0.263	27.394	2.832
	cloud	2.009	0.277	25.354	2.901
Camera 2	bare-metal	2.639	0.239	13.550	3.893
	fog	2.122	0.224	21.809	2.889
	cloud	2.119	0.280	22.533	2.955
Lidar	bare-metal	1.631	0.257	26.339	3.145
	fog	1.344	0.301	11.800	2.203
	cloud	33.567	21.021	51.892	6.129

3) Message Transmission Time: The message transmission time between the camera and the detection node is presented in Table II. The results of the cloud setup show almost no difference compared to the fog setup, as the communication between the camera and the PC is entirely local. A notable discovery from the results was that the mean image transmission time in a fog setup is unexpectedly lower than in a bare-metal configuration. Specifically, transmission times from the camera in the fog environment are 23% shorter on average compared to those in the bare-metal setup. The higher mean transmission time in the bare-metal setup is due to increased variability, indicated by the higher standard deviation. More instances of longer transmission times contribute to the overall higher mean, suggesting that fog/cloud setups provide a more stable and efficient network environment for transmitting images. The above-observed improvements can be attributed to several factors inherent to the design and operation of cloud-native, which will be discussed in detail in the IV-E section.

D. Multi-Robot Experiment

This experiment is designed to assess the efficiency of a fog setup in orchestrating multiple robots, particularly in scenarios that require real-time and wireless communication capabilities. Additionally, we perform the overall resource consumption and latency evaluations under this setup.

1) Startup Time: The startup time of this multi-robot setup is depicted in Fig. 7(d)–(f). The Franka robot's startup time remains consistent with that of previous experiments. However, all the rest of the applications except Lidar under the fog architecture show slightly longer startup times than those under bare-metal conditions. Overall, the total startup time is 14% slower than the bare-metal. As for the cloud setup, all the applications launch slower than the bare-metal, and the total startup time is 49% slower than the bare-metal.

Several factors contribute to this worse result compared to previous experiments. First, the limited processing power of Turtlebot's onboard PC (Raspberry Pi) enlarges the overhead from containerization and Kubernetes, impacting performance. Second, it takes longer for k3s to establish connections with the Turtlebot over a wireless network, while this time is not accounted for in the bare-metal setup. Shown in Fig. 7(e) and (f), the vehicle application, indicating the launch of the Turtlebot,

TABLE III
OVERALL RESOURCE UTILIZATION ACROSS DIFFERENT SETUPS

Metrics	bare (I)	bare (II)	k3s (I)	k3s (II)	k3s (III)
CPU	40.05%	62.58%	48.43%	64.18%	59.69%
Memory	26.63%	40.07%	29.51%	46.74	44.55%
GPU	9.12%	8.99%	9.05%	9.09%	9.17%

(I) and (II) denotes experiment I andII. (III) indicates experiment II without microservice.

TABLE IV
COMPARISON BETWEEN DIFFERENT LATENCY, STD. DENOTES THE STANDARD DEVIATION

Setup	bare-metal		fog (none)		fog (low)		fog (high)	
	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
Camera(ms)	2.572	3.838	2.030	2.861	2.987	2.903	12.415	4.544
OnT(ms)	0.493	0.590	0.591	0.785	1.505	0.791	10.799	2.008
Startup(s)	4.827	0.141	4.602	0.090	4.691	0.101	5.322	0.132

Fog (low) and *fog (high)* represent the latency simulated for fog servers with wired and wireless connections, respectively.

is started 2.9 s and 3.6 s after the command input in the fog and cloud setups, respectively. This means that on the Raspberry Pi, Kubernetes takes this amount of time to assign deployments and create containers. This discrepancy reasonably accounts for the superior performance of the bare-metal setup compared to k3s.

2) Message Transmission Time: Since the Turtlebot operates via a wireless connection, the data transmission time across wireless devices is particularly intriguing. The experiment results are detailed in Table II. An initial observation from the table is that, despite the Lidar data being considerably smaller than image data, both types require similar transportation times. The network connection type and whether the transmission is across devices can influence data transport performance. Consistent with earlier findings, the performance of the cloud setup remains significantly worse compared to the other two configurations due to the increased latency. Similar to the image results, the fog setup has a lower mean transport time and, at the same time, a lower standard deviation. This indicates that fog setup offers superior data transport performance across wired and wireless connections, regardless of message type.

3) Resource Consumption: To assess the impact of containerization and microservices on resource utilization, we record CPU, memory, and GPU usage throughout the experiments, as summarized in Table III. K3s(I) represents Experiment I, K3s(II) represents Experiment II, and K3s(III) represents a scenario where each PC runs a single container without microservices. As shown in table, the use of microservices and containerization increases CPU and memory utilization, with greater resource consumption as the system becomes more decomposed. However, this increase is not a significant concern due to the sufficient capacity of the fog server. Additionally, applications on the onboard PC can be redeployed to the fog server if necessary. Notably, GPU utilization remains consistent across all experiments.

4) Impact of Latency: We conduct additional experiments to examine the effects of multiple fog servers connected via wired Ethernet or wireless network such as Wi-Fi or 5G, each with different latencies. Using the *tc qdisc* package, we simulate latency for wired connections (fog low) with 1 ms and 0.5 ms jitter and for wireless connections (fog high) with 10 ms and 3 ms jitter. As shown in Table IV, low latency (1 ms) has negligible impact on startup time, with only minor increases in message transmission and execution times during online training. In contrast, high latency (10 ms) extends startup time and reduces message and training frequencies below 100 Hz, failing to meet CBF's real-time requirements and causing system instability. These results suggest that network with guaranteed latency that real-time applications require are essential for stable and efficient performance when using multiple fog servers in a microservice-based fog robotic architecture.

E. Discussion

Some of the previously presented results draw counter-intuitive conclusions. We observed that the fog setup launched faster than bare-metal in the first experiment, where there was only one Franka robot. This is surprising because containerization typically introduces additional overhead compared to bare-metal systems, as deployment assignment and container creation consume extra time. The experiments are repeated 20 times to ensure reliable output, and based on the existing results, we found some explanations for this. First, the onboard PC for the Franka robot is a relatively powerful platform compared to the Raspberry Pi. Running a Kubernetes agent and creating five containers simultaneously does not introduce significant overhead. Second, the integration of an enhanced microservice architecture contributes to the improvement. We divide the robot's control modules and obstacle avoidance algorithms into smaller, individual components under the fog setup. This results in each application container encompassing fewer ROS nodes. According to our previous work [20], ROS performance significantly decreases when the number of nodes becomes too large. The Franka robot setup in our experiment is a relatively complex robotic system with a total of 32 nodes. The efficiency gained from having fewer nodes in each application outweighs the overhead introduced by containers and Kubernetes, resulting in faster startup times. However, this improvement is not universal across all robotic setups. In the second experiment, factors such as the lower performance of the Raspberry Pi, the more straightforward Turtlebot module, and the poorer network conditions contribute to longer startup times.

Furthermore, we observed a marked improvement in the transmission time of messages between nodes within the fog architecture. Our analysis suggests that this efficiency can be attributed to the secure, isolated environments created by containerization on the host system. An essential factor in this discussion is the role of cgroup scheduling and core task assignment within the Linux Completely Fair Scheduler (CFS) [21]. Kubernetes and Docker leverage cgroups to manage container workloads efficiently. The CFS allocates resources to cgroups based on their CPU shares (cshares), determining how resources are distributed among containers. One major benefit of this system is the isolation it provides. It prevents isolated processes from being influenced by external processes and stops any single process from monopolizing available resources. Without the containerization layer, processes in a native system can unintentionally interfere with each other, impacting performance. This isolation, combined with the balanced resource allocation enabled by our design space exploration method, contributes to the fog architecture's more efficient and stable performance. In summary, our experiments validate the microservice-based cloud-native fog robotic architecture's feasibility and scalability. They reveal the architecture's potential to enhance performance in startup time and message transport under specific conditions.

V. Conclusion

This paper provides a solution for managing complex cloud-native fog robotic systems by incorporating an enhanced microservice architecture and containerization. This approach transforms traditional monolithic robotic systems into smaller, independent, and more manageable segments, simplifying tasks like modification and maintenance while improving the overall efficiency and flexibility of the system. To address the increased complexity in resource allocation and management, we propose a modeling strategy based on MBSE for system abstraction and design space exploration. The MBSE approach provides several key advantages, such as increased flexibility, simplified system modifications, and seamless hardware migration. However, its implementation requires a substantial initial investment, along with extended solving times as system complexity increases. Despite these challenges, the long-term benefits, including enhanced efficiency and reduced need for manual reconfiguration, outweigh these initial hurdles, making MBSE a valuable strategy in complex system development. In addition, we conducted thorough performance assessments of this architecture compared to traditional bare-metal setups and cloud deployments in real-time scenarios. The fog setup demonstrated enhanced flexibility, agility, and improved performance, with faster startup times and message transmission speeds across both wired and wireless networks. Moreover, the architecture provides a more stable operating environment, as evidenced by reduced variability in all evaluation metrics. Our future work will focus on migrating from ROS1 to ROS2 and incorporating latency considerations into our modeling process.

References

[1] M. Groshev, G. Baldoni, L. Cominardi, A. de la Oliva, and R. Gazda, “Edge robotics: Are we ready? An experimental evaluation of current vision and future directions,” *Digit. Commun. Netw.*, vol. 9, no. 1, pp. 166–174, 2023.

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, and L. Safina, “Microservices: Yesterday, today, and tomorrow,” in *Present and Ulterior Software Engineering*, Berlin, Germany: Springer, 2017, pp. 195–216.

[3] V. Velepucha and P. Flores, “A survey on microservices architecture: Principles, patterns and migration challenges,” *IEEE Access*, vol. 11, pp. 88339–88358, 2023.

[4] G. Hu, W. P. Tay, and Y. Wen, “Cloud robotics: Architecture, challenges and applications,” *IEEE Netw.*, vol. 26, no. 3, pp. 21–28, May/Jun. 2012.

[5] M. Waibel et al., “RoboEarth,” *IEEE Robot. Autom. Mag.*, vol. 18, no. 2, pp. 69–82, Jun. 2011.

[6] R. Doriya, P. Chakraborty, and G. C. Nandi, “‘Robot-Cloud’: A framework to assist heterogeneous low cost robots,” in *Proc. IEEE Int. Conf. Commun. Inf. Comput. Technol.*, 2012, pp. 1–5.

[7] N. Tian et al., “A fog robotic system for dynamic visual servoing,” in *Proc. IEEE Int. Conf. Robot. Automat.*, 2019, pp. 1982–1988.

[8] A. K. Tanwani, N. Mor, J. Kubiatowicz, J. E. Gonzalez, and K. Goldberg, “A fog robotics approach to deep robot learning: Application to object recognition and grasp planning in surface decluttering,” in *Proc. IEEE Int. Conf. Robot. Automat.*, 2019, pp. 4559–4566.

[9] J. Ichnowski et al., “FogROS2: An adaptive platform for cloud and fog robotics using ROS 2,” in *Proc. IEEE Int. Conf. Robot. Automat.*, 2023, pp. 5493–5500.

[10] T. Cerny, M. J. Donahoo, and M. Trnka, “Contextual understanding of microservice architecture: Current and future directions,” *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 29–45, Jan. 2018.

[11] L. Roda-Sanchez et al., “Cloud–edge microservices architecture and service orchestration: An integral solution for a real-world deployment experience,” *Internet Things*, vol. 22, 2023, Art. no. 100777.

[12] V. Singh and S. K. Peddoju, “Container-based microservice architecture for cloud applications,” in *Proc. IEEE Int. Conf. Comput. Commun. Autom.*, 2017, pp. 847–852.

[13] L. Wen, M. Rickert, F. Pan, J. Lin, and A. Knoll, “Bare-metal vs. hypervisors and containers: Performance evaluation of virtualization technologies for software-defined vehicles,” in *Proc. IEEE Intell. Veh. Symp.*, 2023, pp. 1–8.

[14] C. Xia, Y. Zhang, L. Wang, S. Coleman, and Y. Liu, “Microservice-based cloud robotics system for intelligent space,” *Robot. Auton. Syst.*, vol. 110, pp. 139–150, 2018.

[15] U. Pohlmann and M. Hüwe, “Model-driven allocation engineering: Specifying and solving constraints based on the example of automotive systems,” *Autom. Softw. Eng.*, vol. 26, pp. 315–378, 2019.

[16] I. Al-Azzoni, J. Blank, and N. Petrović, “A model-driven approach for solving the software component allocation problem,” *Algorithms*, vol. 14, no. 12, p. 354, 2021.

[17] F. Pan, J. Lin, M. Rickert, and A. Knoll, “Automated design space exploration for resource allocation in software-defined vehicles,” in *Proc. IEEE Intell. Veh. Symp.*, 2023, pp. 1–8.

[18] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada, “Control barrier functions: Theory and applications,” in *Proc. 18th IEEE Eur. Control Conf.*, 2019, pp. 3420–3431.

[19] Y. Zhang et al., “Online efficient safety-critical control for mobile robots in unknown dynamic multi-obstacle environments,” in *Proc. IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, ADNEC, Abu Dhabi, 2024 In Press, doi: 10.48550/arXiv.2402.16449.

[20] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, “OPC UA versus ROS, DDS, and MQTT: Performance evaluation of industry 4.0 protocols,” in *Proc. IEEE Int. Conf. Ind. Technol.*, 2019, pp. 955–962.

[21] T. Combe, A. Martin, and R. Di Pietro, “To docker or not to docker: A security perspective,” *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 54–62, Sep./Oct. 2016.