IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 13 February 2024, accepted 2 March 2024, date of publication 11 March 2024, date of current version 22 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3376570

RESEARCH ARTICLE

Performance Analysis of Container Technologies for Computer Vision Applications on Edge Devices

OSAMAH I. ALQAISI[1,3], ALI ŞAMAN TOSUN[2], AND TURGAY KORKMAZ[1]
[1]Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX 78249, USA
[2]Department of Mathematics and Computer Science, The University of North Carolina at Pembroke, Pembroke, NC 28372, USA
[3]College of Computing and Information Technology, University of Tabuk, Tabuk 71491, Saudi Arabia

Corresponding author: Osamah I. Alqaisi (osamah.alqaisi@my.utsa.edu)

Research was sponsored by the Army Research Office and was accomplished under Grant Number W911NF-23-1-0187. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, not withstanding any copyright notation herein.

ABSTRACT In the dynamic realm of technology, various container technologies offer efficient deployment and resource utilization in edge devices. However, limited research has explored how various container technologies perform in specific domains. In response, this paper addresses this gap by evaluating container technologies like RunC, LXC, Containerd, Docker, Podman, and Singularity in OpenCV-based computer vision applications on ARM-based edge devices. Results show comparable performance between containerized and non-containerized applications. Containerd excels in memory reading, with both Containerd and LXC efficient in wired image reception, while Singularity and Containerd lead in wireless image reception. Despite Docker's slower memory reading, its consistently faster processing time positions it as a competitive option. Overall, Docker demonstrates superior efficiency for computer vision applications on ARM-based edge devices. These insights contribute to bridge the existing gap in integrating containers into IoT and ARM-based edge computing scenarios.

INDEX TERMS Container technology, Docker, LXC, Podman, Singularity, Containerd, RunC, edge computing, Raspberry Pi, computer vision.

I. INTRODUCTION

Utilizing the edge computing paradigm for computer vision applications is crucial to meet real-time processing needs, reduce latency, and improve the efficiency of video data analysis. However, this presents challenges, primarily due to the limited computational resources of edge devices. The diverse landscape of edge computing systems poses a critical challenge, leading to compatibility issues for developers working on various devices. Ensuring adaptability and uniformity in programming interfaces becomes essential for seamless execution of computer vision applications across different hardware architectures. The intricate architectures of edge devices add complexity to device management. Furthermore, the distributed nature of edge devices and the multitude of possible network configurations increase the overall costs associated with maintaining a robust and efficient computer vision infrastructure at the edge [1], [2].

In these regards, container technology stands out and offers powerful solutions to overcome these limitations, paving the way for the efficient and scalable deployment of computer vision applications across diverse domains and unlocking the full potential of edge computing.

Container technology has emerged as a transformative form of virtualization, revolutionizing software development, particularly in the context of the Internet of Things (IoT) and edge computing. Its widespread adoption in the field of software development, specifically for edge computing applications and IoT, has propelled it into the limelight. Containers facilitate the compartmentalization of applications into self-contained units to avoid the need for an entire operating system. This characteristic makes containers

The associate editor coordinating the review of this manuscript and approving it for publication was Hadi Tabatabaee Malazi.

lightweight, portable, and conducive to easy application deployment across a wide array of devices. In particular, a key strength of container technology emerges in the deployment of applications within edge computing settings. By extricating applications from the underlying operating system, containers mitigate integration complications and lessen the risk of inconsistencies. This proves especially advantageous for IoT applications, where containerized computer vision applications can find a home even in resource-constrained devices. The streamlined design of containers allows edge devices to efficiently juggle multiple applications on identical hardware. Notably, containers improve security by isolating applications, thwarting attacker attempts to infiltrate the entire system. This isolation streamlines updates and patches, reducing the vulnerability window. In summary, container technology proves invaluable for developers navigating the landscapes of edge computing and IoT. The ability of containers to isolate applications from the core operating system and eliminate extraneous elements makes them an asset in the treatment of various challenges within these domains [3], [4], [5].

The current container landscape is populated by a multitude of container runtimes/engines, each distinguished by its unique architecture. However, ongoing efforts are being made to address fragmentation within this domain by promoting the adoption of standards. These standards are gaining traction, particularly among prominent projects such as Docker, which is helping to alleviate the divergence issue. Despite these efforts, the distinction between container technologies of applications and operating systems (OS) remains a defining factor within the container landscape [6]. Furthermore, the container base image and size contribute to overall performance, introducing additional considerations in the containerization ecosystem [7].

This article focuses on evaluating and comparing major container runtimes/engines in edge computing environments. Its goal is to assist users in selecting the most suitable container technology for CPU-based computer vision applications based on the OpenCV library, with the aim of optimizing the performance of containerized computer vision applications while supporting multiple IoT sensors. By employing CPU-based computer vision algorithms like Haar Cascades, HOG, and Tiny-CNN, the research evaluates different container technologies to provide comprehensive insights into their strengths and weaknesses in Computer Vision Applications. Its primary objective is to quantify the performance impact of container virtualization technology compared to native execution, providing assessment results to facilitate informed decision-making in selecting container technologies for computer vision applications in edge computing environments. Contributions of the paper are as follows:

- Review of prominent container technologies, Docker, LXC, Containerd, RunC, Podman, and Singularity, specifically tailored for ARM-based edge devices.
- Analysis of the process of building an OpenCV Docker container on different container runtime/engine platforms, offering insights into compatibility and performance implications.
- Performance analysis covering key performance indicators such as receiving time, processing time, memory usage, and CPU usage for a comprehensive assessment of each container's capabilities.
- Identify the strengths and limitations of each container technology, offering a guide for developers to choose the best for computer vision on ARM-based edge devices.

The paper is structured as follows. Section II presents a review of related work, while Section III introduces the proposed scheme. The results are presented in Section IV. Section V discusses the important aspects of the results. The paper concludes in section VI and discusses future works.

II. RELATED WORK

In this section, we discuss related work on container technology, performance of container runtimes/engines, and computer vision applications with container technology.

A. CONTAINER TECHNOLOGY

Container technology offers isolation within a single host, different from virtual machines. Containers provide benefits such as lightweight deployment, resource efficiency, and version control, which makes them suitable for microservices. They are utilized in IoT services [8], [9], smart cars, fog computing [10], and service meshes [11], [12]. The major organizations, including Amazon, Spotify and Netflix, have established containers as the standard for cloud deployment [13], [14]. Various runtimes/engines like Docker, LXC, RunC, Singularity, Podman, and Containerd exist, with Docker being the most popular. The Open Container Initiative (OCI) standardizes format and runtime requirements, ensuring portability across runtimes. However, the choice of runtime/engine significantly impacts container technology's features and performance [15].

- **RunC**: following the Open Containers Initiative (OCI) specification, is a lightweight and highly portable container runtime widely employed in container orchestration platforms. As a command-line tool, it manages containers by providing key functionalities such as creation, launch, stop, and deletion. RunC supports Linux namespaces and incorporates various Linux security features to prioritize robust isolation between containers and the host operating system. Emphasizing security, it ensures independent file systems, network stacks, and process tables for each container. With features such as live migration and compatibility with OCI specifications, RunC is a versatile choice for deploying containers in diverse environments [16], [17].
- **Linux Containers**: LXC is a lightweight OS-level virtualization technique that enables the execution of multiple isolated Linux systems on a single host that share a common Linux kernel. Known for its resource

efficiency, LXC allows multiple applications to run on a single server with lower resource consumption than individual virtual machines. Prioritizing security, LXC isolates each container from others and the host OS. Its flexibility extends to various use cases, from hosting multiple applications to creating isolated development environments, implementing CI/CD pipelines, and deploying applications. Favored by major companies such as Google, Amazon, and Microsoft, LXC is renowned for its efficiency, security, flexibility, and robust community support [18], [19].

- **Containerd**: is an industry-standard container runtime/engine, focusing on simplicity, robustness, and portability while managing the full container lifecycle, including image transfer, storage, container execution, and supervision. Containerd is designed to be embedded in broader systems rather than for direct development or end-user use. Key advantages encompass its lightweight and efficient nature, strong container isolation for enhanced security, cross-platform support on Linux and Windows [20], [21].
- **Docker**: is a runtime that manages container lifecycles, from creation to shut-down, ensuring robust isolation. Built on the OCI standard, it is versatile, running containers from diverse sources like Docker Hub. Compatible with Linux, Windows, and macOS, Docker is widely used in Kubernetes. With minimal resource usage, it facilitates easy project transfer between devices, providing isolation and segregation [22]. DockerHub [23] enables the sharing of containerized applications, making it suitable for ARM platforms, as well as scanning and security tools [24]. In general, Docker stands out for its ease of use, security, and community support on ARM platforms.
- **Singularity**: is a powerful container runtime/engine designed for high-performance computing (HPC) clusters. It offers a unique feature, allowing image mounting without requiring root access, making it stand out in the container landscape. Singularity builds on Linux containers and combines software stacks into a single configuration file for versatile container creation and deployment on various platforms. Its runtime/engine balances integration and separation, enabling smooth data exchange with the host system and utilizing high-speed interconnects and GPUs. Additionally, Singularity streamlines the migration of Docker containers into its environment, emphasizing user convenience and compatibility. With support for ARM64 architecture, Singularity caters to edge computing needs, making it a secure and adaptable containerization solution for HPC, security, and edge environments [25].
- **Podman**: introduces a revolutionary daemon-less approach to manage and run OCI containers on Linux systems. It offers flexibility, allowing containers to operate under root or rootless privileges, and enhancing security through isolation and user privilege management. Podman coexists seamlessly with Docker, offering compatibility by aliasing the Docker CLI commands with Podman, easing the transition between the two environments. It provides supplementary CLI tools and enables the effortless migration of Docker containers, enhancing its versatility and ensuring a smooth switch between containerization platforms. With strong compatibility, security features, and ongoing development, Podman is a compelling choice for ARM platforms, offering efficient, secure, and forward-looking containerization solutions [26].

B. PERFORMANCE OF CONTAINER TECHNOLOGY

Research on the evaluation of container runtimes is limited, and the majority of existing studies focus on comparing traditional virtual machines to container technology. However, these studies typically examine the differences among container technologies in a general context, lacking a specific focus on particular domains or applications for comparative analysis. In this crucial field of researching container technology performance, studies often involve comparing container efficiency using synthetic tools. This comparison includes analysis of CPU, RAM, and network efficiencies, commonly achieved through computations of prime numbers or accessing MySQL databases.

In a study by Cailliau et al. [6], container runtimes/engines such as Docker, Rkt, and LXC were evaluated on ARM architecture. Notably, Docker and LXC exhibited comparable performance across RAM, CPU, and network metrics, while Rkt's performance was more constrained. Kovacs [27] assessed Docker, LXC, and Singularity containers on Huawei CH121, focusing on CPU and network performance. Another study by Espe et al. [28] evaluated Containerd, runC, and CRI-O on CPU, memory, and I/O performance. Velp et al. [7], Li [29], and Wang [30] conducted comparisons between virtual machines and containers (Docker, Podman, and LXC) in Intel architecture, examining CPU, memory, read and write I/O, network, and database read and write. The consistent findings of these investigations highlight Docker's superior performance among containers, surpassing that of virtual machines.

In the realm of ARM edge devices, Marques et al. [31] conducted a performance assessment of the LXC runtime for virtualization on devices such as the Raspberry Pi 2. Fernandez Blanco et al. [32] evaluated Docker and LXC in automotive systems using Raspberry Pi 2, 3, and 4 boards, consistently favoring Docker for superior performance with minimal overhead, followed by LXC. Jing et al. [33] assessed Docker and Containerd runtimes/engines on Raspberry Pi 3 and 4 as edge devices. Morabito [4], [34] performed a comprehensive performance evaluation on five ARM-based single-board processors: Raspberry Pi 2 Model B, Raspberry Pi 3 Model B, Odroid C1+, Odroid C2 and Odroid XU4, using Docker containers exclusively. Acharya et al. [35] conducted a study using Docker on ARMv8 and x86 architectures. Raho et al. [36] evaluated Docker on the Arndale ARMv7

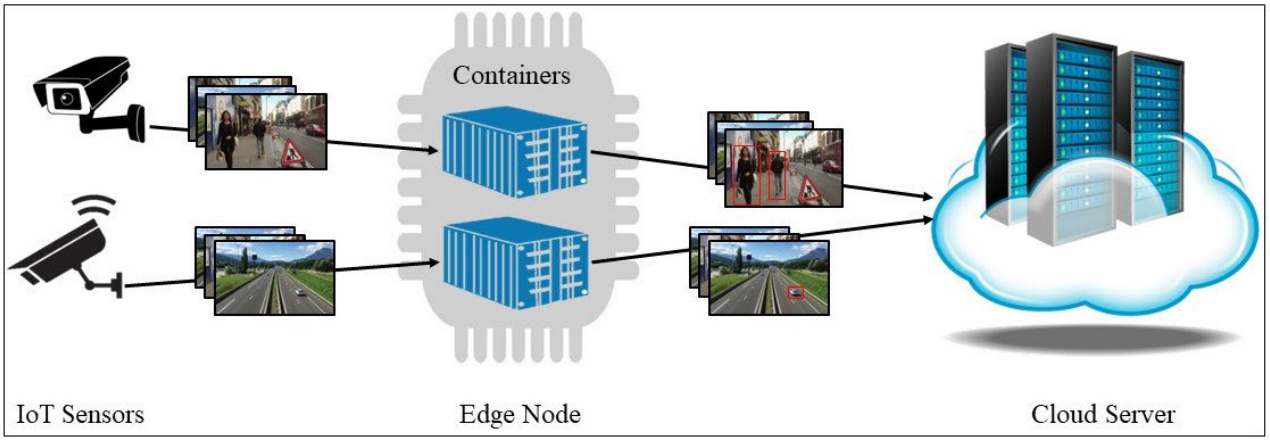


FIGURE 1. Overview of container-based edge system.

Board. All of these studies consistently highlight the commendable performance of container technology over virtual machines, particularly Docker and LXC.

C. CONTAINERIZED COMPUTER VISION APPLICATIONS

Various container management platforms, such as Apache OpenWhisk [37], OpenFaaS [38], Kubeless [39], and Docker Swarm [40], are available. Alabbas et al. [41] evaluate the performance of Apache OpenWhisk with Docker containers in an edge-cloud context, specifically exploring a video analysis application. Their study, conducted on Raspberry Pi 4 edge nodes, reveals insights into OpenWhisk's cold activation latency and its sensitivity to memory and input size.

Seisa et al. [42] compare two edge architectures for orchestrating a UAV's path using ROS: one utilizing Docker containers and the other employing Kubernetes. Simulations assess factors like time delays, travel durations, execution times, and CPU utilization, revealing both approaches as efficient. Docker may be more suitable for smaller, less complex projects, leading to an application-dependent choice between the architectures.

Alqaisi et al. [43] implement lightweight containers for computer vision in edge IoT devices, creating containers for C++ and Python with OpenCV. Tested in Haar Cascade, HOG, and Yolov3 scenarios on a Raspberry Pi 4, the study emphasizes Docker's benefits like resource optimization and mobility. Best practices, such as minimizing container size and managing concurrent containers, significantly enhance performance.

The analysis of the related work section indicates that the previous studies discussed therein did not evaluate the effectiveness of containerized computer vision applications for Edge computing, considering the particular objective of these applications to achieve swift responses. As a result, we propose to perform an evaluation specifically focused on containerized computer vision deployed on edge devices.

III. SYSTEM ARCHITECTURE AND CONFIGURATION

In this article, a container-based edge system has been established on an ARM device at the edge. The images are received by a container runtime/engine, followed by the analysis of these images and the execution of necessary actions. Subsequently, the analyzed images are transmitted to the cloud system for storage and further processing. The proposed system is conceptually depicted in Figure 1. However, for the purpose of this experimental set-up, conventional transmission to the cloud system has been excluded. Instead, the output images are stored within the container's memory. More details about this experimental setup, including three scenarios and five applications, will be elaborated on. Additionally, specific time intervals between the reading/receiving of images will be expounded.

TABLE 1. Raspberry Pi 4 specifications.

Specification	Description
Processor	1.5 GHz 64-bit quad-core ARM Cortex-A53
Memory	8 GB LPDDR4 with on-die ECC
GPU	Broadcom VideoCore VI 500 MHz
Connectivity	2.4 GHz &5.0 GHz 802.11ac Wi-Fi Gigabit Ethernet 2 × USB 3.0 ports 2 × USB 2.0 ports.
SD card support	Micro SD card slot for loading operating system and data storage
Operating System	Raspbian 32 bit, a free OS based on Debian

A. EDGE NODE CONFIGURATION

The edge system and the IoT sensor used in this study are based on a Raspberry Pi 4 (RPi 4). As shown in Table 1, the specifications of the device include a 1.5 GHz 64-bit quad-core ARM Cortex-A53 CPU, 802.11ac Wi-Fi and gigabit Ethernet connectivity, 8 GB of RAM, and a Broadcom VideoCore VI 500 MHz GPU. The operating system used on RPi 4 operates on Raspbian 32-bit, an open-source operating system rooted in Debian version 11 (bullseye) [44].

The RPi 4 is equipped with a range of container technologies, each with its specific version. It is worth noting that each of these container runtimes/engines is individually installed on the system, without the concurrent installation of others. These containers encompass the OpenCV library version 4.3.0, serve as the foundation for running various

computer vision applications, and NumPy library version 1.22.3. The implementation of these applications is carried out using the Python version 3.10.4 programming language.

The NetGear AC1000 Wi-Fi Router (R6080) is used for network connectivity, which supports 802.11ac wireless technology and incorporates four 10/100 Mbps LAN ports [45].

B. CONTAINER CONFIGURATION

In this experiment, the edge system is used to accommodate containers through RunC, LXC, Containerd, Docker, Podman, and Singularity container systems. The versions used for each container are LXC 5.18, RunC 1.1.0+dev, Containerd 1.4.13-ds1, Docker 20.10.5+dfsg1, Singularity 3.8.0, and Podman 3.0.1.

All container runtimes/engines utilize the same container image, which is created using a Dockerfile. This Docker image is constructed on the Ubuntu 20.04.3 LTS operating system, incorporating essential packages and libraries such as Python, Pip, NumPy, and OpenCV, all installed from their binary releases. The Dockerfile concludes by eliminating any superfluous packages to ensure optimal efficiency. This container image can be found on Docker Hub under the name *oalqaisi/cv_python:latest*.

The Podman container image is pulled directly from Docker Hub using the command *podman pull docker.io/cv_python:latest*. Similarly, the Containerd container image is fetched from the Docker Hub using the command *ctr image pull docker.io/oalqaisi/cv_python:latest*.

In the case of the Singularity container image, it is constructed from the Docker image found on Docker Hub using the command *sudo singularity build ≪name≫.simg docker://oalqaisi/cv_python:latest*.

In the case of LXC and RunC runtimes/engines, the LXC container image is created from the Docker image on the Docker Hub using the command *sudo lxc-create ≪name≫ -t oci – –url docker://oalqaisi/cv_python:latest*. In the RunC runtime/engine, the process involves extracting the root file system from the Docker container image *oalqaisi/cv_python:latest* and then generating a configuration file using *runc spec*. Following container image acquisition, the network configuration for both LXC and RunC container images is integrated into the config file, and a bridge between the containers and the host device is manually established. Lastly, a port forwarding rule is added to *iptables* to enable data forwarding to the containers.

C. EXPERIMENT SCENARIOS

This study covers three implementation scenarios, as shown in Figure 2. The first scenario involves an application that reads images from memory, with both the application and the images co-existing within the same container. In this context, applications read an image every second.

The second and third scenarios share a common structure, differing only in the type of connection used for the IoT sensor: a wired connection for the second scenario and a wireless connection for the third. However, the edge device

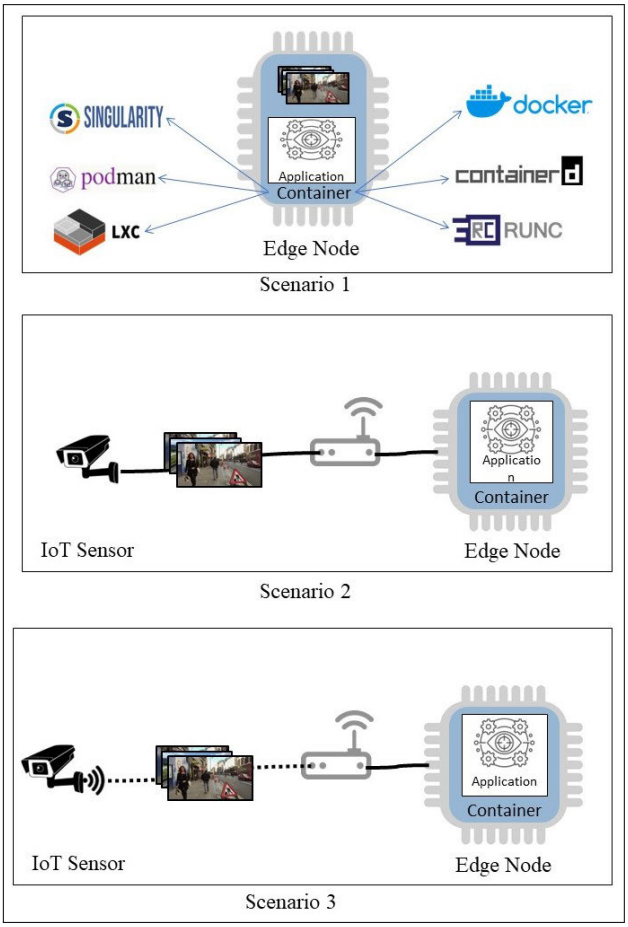


FIGURE 2. Three scenarios in the implementation.

maintains a wired connection to the router in both scenarios. Authentic network scenarios depict real-life scenarios and simulate pragmatic edge computing environments, and previous research has been done on these scenarios. The IoT sensor begins image transmission, concluding each transmission with a unique marker, *##END##*, which signifies the end of an image and prevents any overlap between successive images. This sequence is iterated for each of the 4,000 test images. The IoT sensor terminates the connection with the edge node after transmitting all 4,000 test images. These two scenarios are designed based on observations from prior research [43], which utilized Docker containers, indicating that a wireless connection scenario consumes more resources compared to a wired connection scenario. In this study, these two scenarios are implemented to investigate the variations across different container runtimes/managers.

Within the context of Face Detection, Vehicle Detection, Body Detection, and Object Detection by MobileNet-SSD applications, the IoT sensor transmits an image every second. On the contrary, the Object Detection by YoloTiny application operates at a slightly more relaxed pace, with the sensor dispatching an image every one and a half seconds. Upon receiving an image, an application participates in the detection process and, if it identifies any objects, the output image is stored within the container.

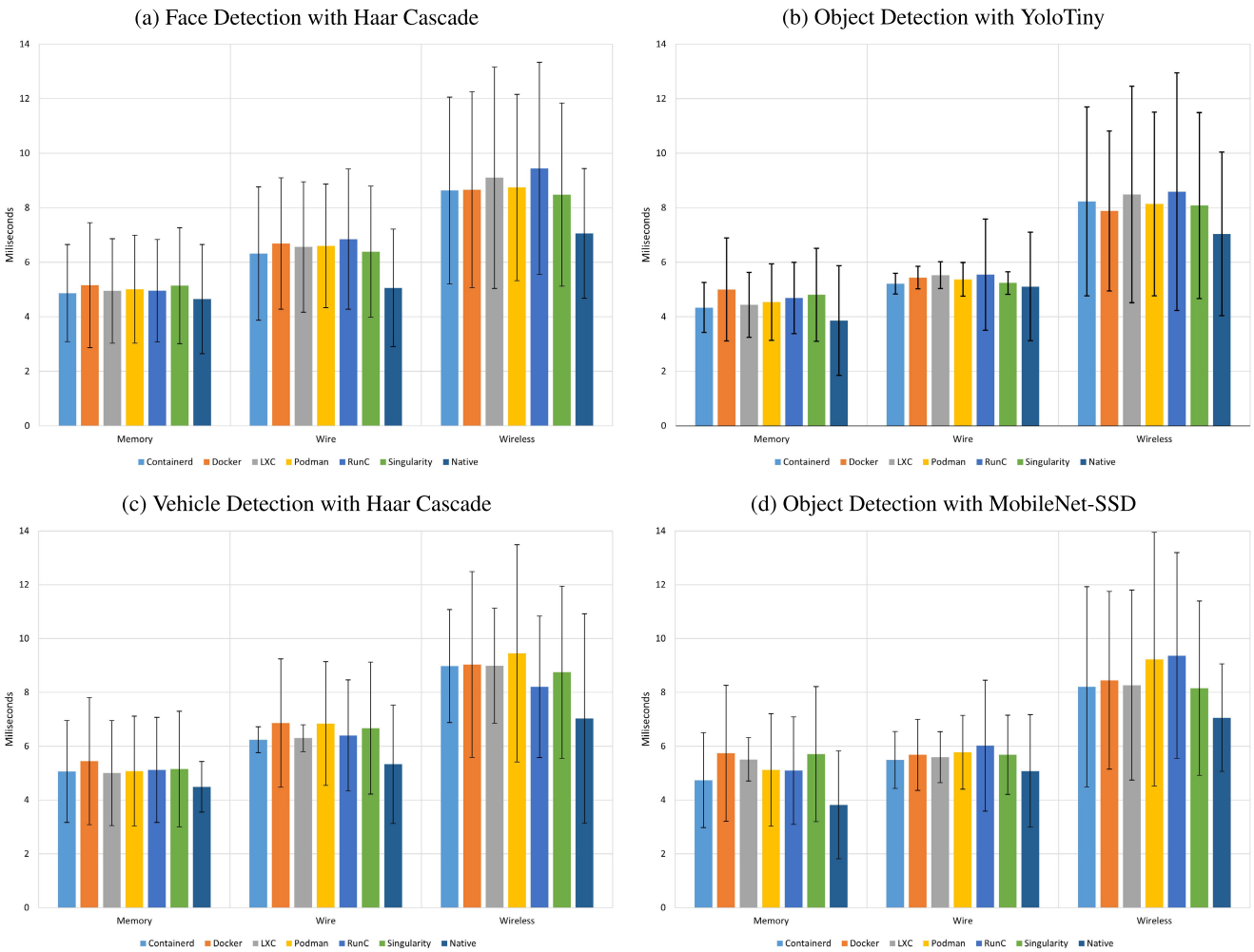


FIGURE 3. Average reading/receiving for haar cascade & CNN apps.

D. APPLICATIONS USED

In this experiment, five different applications are used for a variety of CPU-based computer vision tasks: face detection and vehicle detection applications that implement the Haar Cascades algorithm [46], [47], a body detection application that uses the histogram of Oriented Gradients (HOG) algorithm [48], and two object detection applications that employ the convolutional neural networks (CNN) algorithms of the YoloTiny [49] and MobileNet-SSD [50] algorithms, as shown in Table 2.

These algorithms are often used in edge computing for object detection and deliver exceptional real-time object detection performance even when executed on the CPUs of devices with limited resources, allowing fast and efficient processing of live camera feeds or video streams. Given the real-time data transmission from IoT sensors to edge devices, these algorithms are highly suitable for research scenarios.

The image dataset used in this experiment is drawn from two different videos [51], [52]. Each application is evaluated on a set of 4000 images, all sharing dimensions of 1280 × 720 pixels. These images consist of 1000 unique images, repeated four times. The face and body detection applications share the same set of 4000 images, while the vehicle detection applications utilize a separate image set. The object detection applications draw from a combination of images sourced from both vehicle and face detection applications, with the first 2000 images originating from the vehicle detection application and the last 2000 from the face detection application.

TABLE 2. Application used.

Applications	Files	Size
Face Detection	haarcascade_frontalface_default.xml	940 KB
Vehicle Detection	haar-cascade cars.xml	116 KB
Body Detection	-	-
YOLOTiny v3	yolov3-tiny.cfg yolov3-tiny.weights coco.names	1.87 KB 33.7 MB 625 Bytes
MobileNet-SSD	deploy.prototxt mobilenet_iter_73000.caffemodel	43.6 KB 22.2 MB

E. EVALUATION MATRIX

For the comprehensive evaluation of each application, a series of pertinent metrics has been meticulously gathered, covering a spectrum of performance aspects. These metrics include waiting time, receiving time, processing time, and RAM

utilization. To precisely monitor the resource consumption in terms of CPU and RAM, the capabilities of the *psutil* library have been harnessed, specifically employing version 5.9.5 for these measurements. Utilization of system resources is reported as CPU usage in the form of percentages, and RAM usage quantified in megabytes (MB). It is noteworthy that all the applications under scrutiny benefit from the utilization of pretrained models in the realm of computer vision. These models are seamlessly incorporated using the OpenCV library. Calculating waiting, receiving, and processing times is facilitated by an equation that entails the usage of built-in functions within the OpenCV library, namely the *getTickCount* and *getTickFrequency*. The temporal metrics of waiting, receiving, and processing times are expressed in milliseconds, providing a fine-grained insight into the temporal efficiency of these applications.

IV. EXPERIMENTAL RESULTS

This section presents experimental results for comparison metrics, encompassing read/receive time, image processing time, and resource usage such as CPU and RAM. The measurements cover both average and total times, considering a dataset of 4000 images.

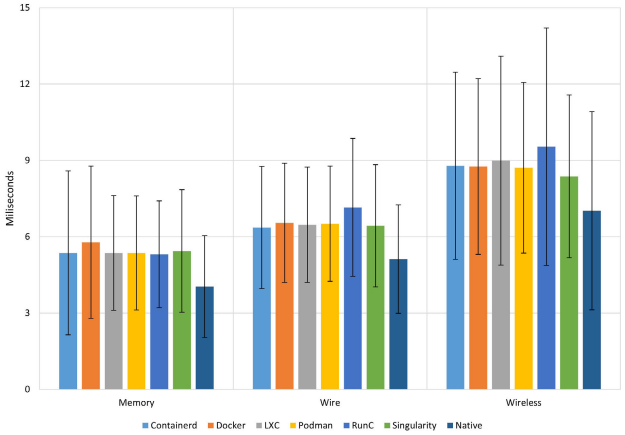


FIGURE 4. Average reading/receiving for HOG app.

A. READING/RECEIVING TIME

The examination delves into the reading or receiving times for all applications, performing experiments with various container runtimes/engines, and native execution on the RPi 4 within the three scenarios.

Examining Figure 3, Figure 4, Figure 5 and Figure 6, which illustrate the average and total reading of images from memory, as well as the reception of images through wired and wireless connections, reveal notable findings. Containerd emerges as the fastest in reading images from memory across most applications, although LXC and RunC exhibit comparable performance in certain applications, sometimes even surpassing Containerd. In contrast, Docker consistently demonstrates the slowest reading performance in most applications.

Regarding the reception of images through wired or wireless networks, performance variability is evident across applications, making it challenging to definitively identify the superior container technology. However, Containerd, Docker, Singularity, and LXC demonstrate notable speed in different applications. In particular, RunC emerges as the slowest container in receiving images, potentially attributed to manual network configuration for containers. Other container runtimes/engines may be built upon RunC, integrating enhanced networking, storage, and image management features. Additionally, these runtimes/engines may employ optimized network drivers or configurations, resulting in faster data transfer compared to a basic RunC setup.

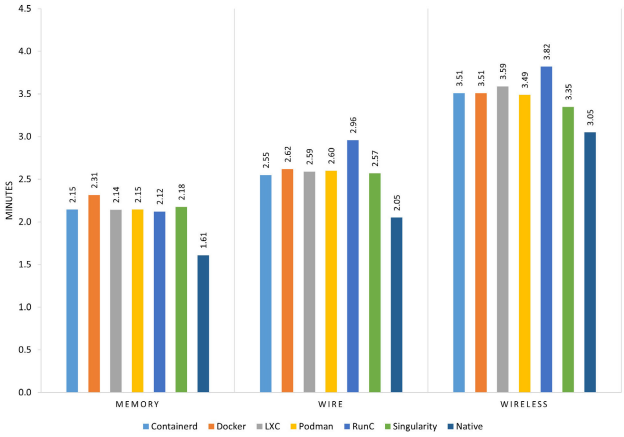


FIGURE 5. Total reading/receiving for HOG app.

Native execution consistently showcases efficiency across all scenarios. Although there are differences, particularly in some applications, these disparities are relatively minor, with average variances ranging from 0.10 to 0.80 milliseconds and total differences spanning from 0.10 to 0.54 seconds.

B. PROCESSING TIME

The experimentation encompasses six container runtimes/engines, together with native execution directly on RPi 4. In this section, a comprehensive summary of the results obtained will be presented, which sheds light on the efficiency and performance of each runtime/engine and the native execution option.

1) OBJECT DETECTION BY YOLOTINY

Figure 7 and Figure8 present comprehensive results for YoloTiny object detection across three scenarios. Docker consistently leads in all scenarios, with average processing times 90.68, 97.01 and 98.16 milliseconds and total processing durations 36.27, 39.11 and 38.94 minutes.

In the first scenario, alternative runtimes, including Singularity, Podman, LXC, Containerd, and RunC, exhibit competitive performance, with total processing times ranging from 36.83 to 38.86 minutes. Native execution on RPi 4 is marginally faster than Docker. Moving to the second scenario, other run-times show average processing times from 97.90 to 103.50 milliseconds and total durations from

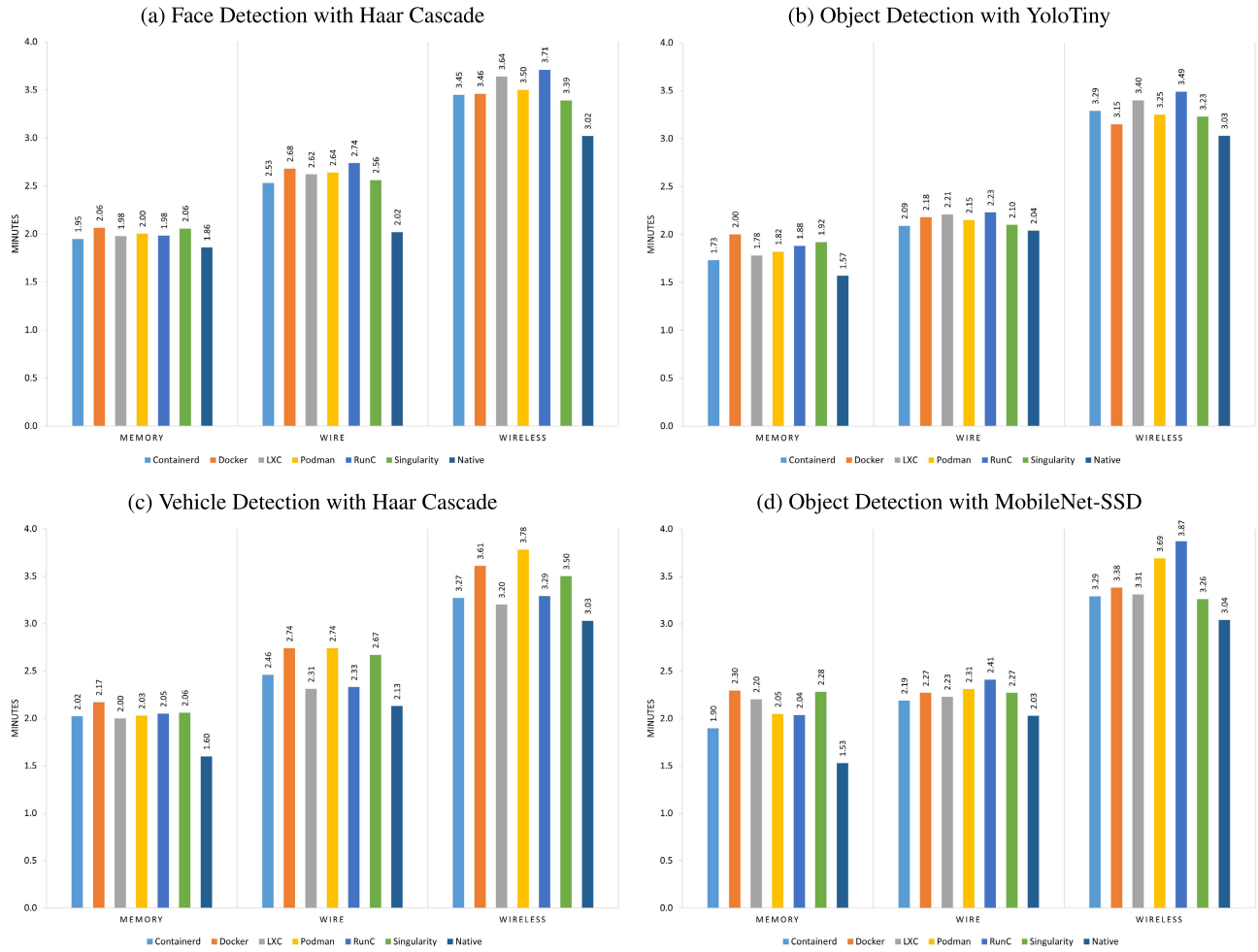


FIGURE 6. Total reading/receiving for haar cascade & CNN apps.

39.16 to 41.40 minutes. RPi 4's native execution surpasses all runtimes, with an average time of 98.16 milliseconds and a total duration of 37.26 minutes. In the wireless scenario, runtimes display average processing times between 98.36 and 100.60 milliseconds, with total durations of 39.34 to 40.24 minutes. RunC is the slowest, with an average time of 104.96 milliseconds and a total duration of 41.99 minutes. RPi 4's native execution remains the fastest, with an average time of 96.41 milliseconds and a total duration of 37.36 minutes.

2) OBJECT DETECTION BY MOBILENET-SSD

Figure 9 and Figure 10 present a detailed analysis of the object detection application using the MobileNet-SSD Algorithm in various container runtimes/engines and native execution on RPi 4. Docker consistently outperforms in all three scenarios, with total processing times of 18.42, 19.59, and 19.85 minutes, respectively.

In the first scenario, alternative runtimes exhibit competitive performance, with total processing times ranging from 18.68 to 19.97 minutes. Containerd is the slowest, recording a total processing duration of 20.38 minutes. Notably, native execution on RPi 4 proves marginally faster, with an average processing time of 45.33 milliseconds and a total processing time of 16.13 minutes.

For the second scenario, other container technologies show average processing times from 49.82 to 53.74 milliseconds and total durations from 19.84 to 21.50 minutes, with Containerd identified as the slowest. Meanwhile, native execution on RPi 4 maintains an average processing time of 49.18 milliseconds and a total processing time of 17.67 minutes.

In the third scenario, container technologies show average processing times between 49.74 and 52.83 milliseconds, with total durations ranging from 19.90 to 21.13 minutes. RunC is the slowest, recording an average processing time of 53.53 milliseconds and a total duration of 21.41 minutes. Once again, native execution on RPi 4 maintains an average processing time of 48.06 milliseconds and a total processing time of 18.02 minutes.

3) BODY DETECTION WITH HOG

Figure 11 and Figure 12 present a concise overview of body detection assessment using the HOG algorithm.

In the first scenario, Singularity excels with an average processing time of 34.42 milliseconds and a total processing

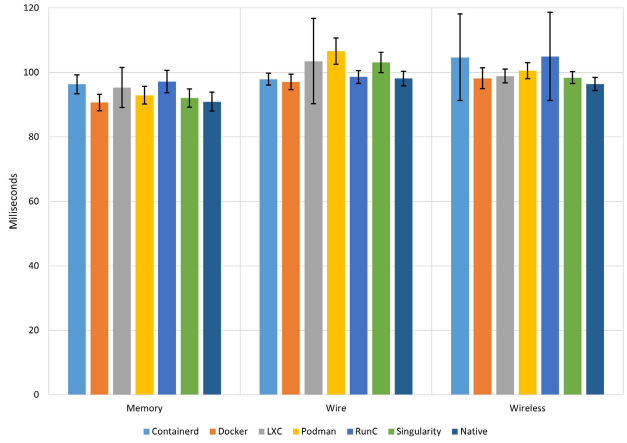


FIGURE 7. YoloTiny - average processing time.

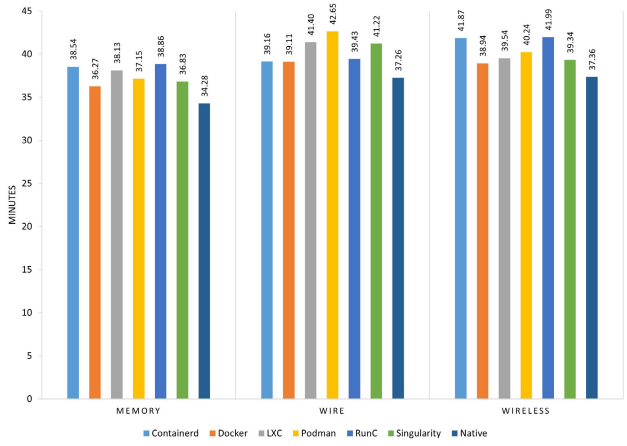


FIGURE 8. YoloTiny - total processing time.

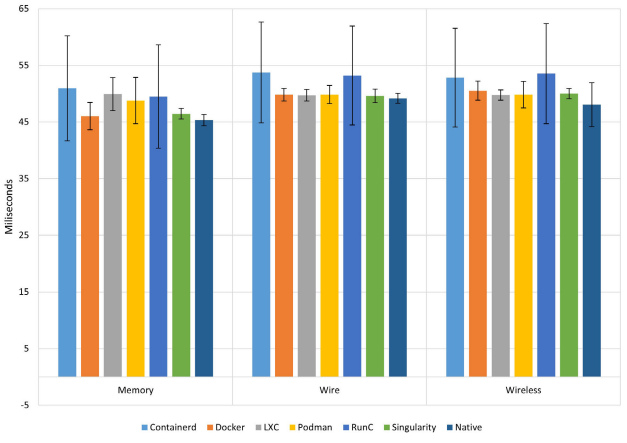


FIGURE 9. MobileNet-SSD - average processing time.

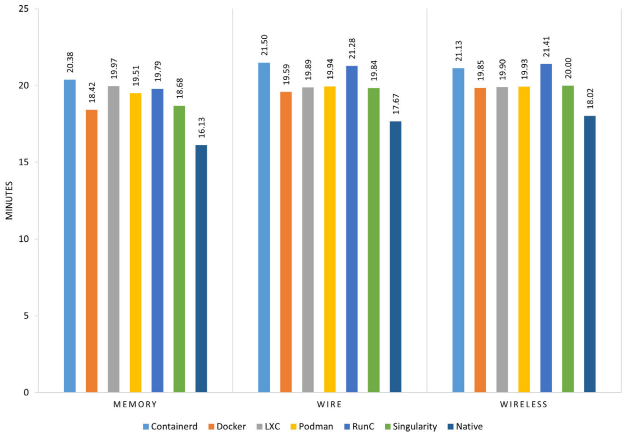


FIGURE 10. MobileNet-SSD - total processing time.

time of 13.77 minutes. Docker closely follows with an average time of 34.49 milliseconds and a total time of 13.80 minutes. RunC, the slowest, records a total processing time of 14.31 minutes. The native execution on RPi 4 demonstrates efficiency with an average time of 24.51 milliseconds and a total time of 9.80 minutes.

For the second scenario, Containerd achieves an average time of 38.06 milliseconds and a total time of 15.22 minutes. Other container technologies range from 15.48 to 15.72 minutes, with RunC being the slowest at 16.10 minutes. The native execution maintains an average time of 27.65 milliseconds and a total time of 11.06 minutes.

In the third scenario, Podman has the fastest total processing time of 15.46 minutes, while RunC has the slowest at 16.25 minutes. RPi 4 native execution maintains a total processing time of 12.36 minutes.

4) FACE DETECTION WITH HAAR CASCADE

Figure 13 and Figure 14 provide insights into face detection using the Haar Cascade Algorithm, highlighting average and total processing times across all three scenarios.

In the first scenario, Docker emerges as the fastest container runtime/engine, with an average processing time of 15.91 milliseconds and a total processing time of 6.36 minutes. Containerd exhibits the slowest total processing time at 6.95 minutes, while native execution on RPi 4 shows a total processing time of 6.56 minutes.

For the second scenario, LXC leads with a total processing time of 7.77 minutes. Containerd, Docker, Singularity, RunC, and Podman follow closely, with total processing times ranging from 7.80 to 8.34 minutes. Native execution on RPi 4 maintains a total processing time of 8.03 minutes.

In the third scenario, Singularity displays an average processing time of 19.67 milliseconds, resulting in a total processing time of 7.87 minutes, making it the fastest among container runtimes/engines. Docker follows, with the other containers trailing behind. The native execution on the RPi 4 directly maintains an average processing time of 19.07 milliseconds, resulting in a total processing time of 7.43 minutes.

5) VEHICLE DETECTION BY HAAR CASCADE

Figure 15 and Figure 16 provide information on face detection using the Haar Cascade Algorithm, highlighting average and total processing times in all three scenarios.

In the first scenario, Docker emerges as the fastest container runtime/engine, with an average processing time of 15.91 milliseconds and a total processing time of

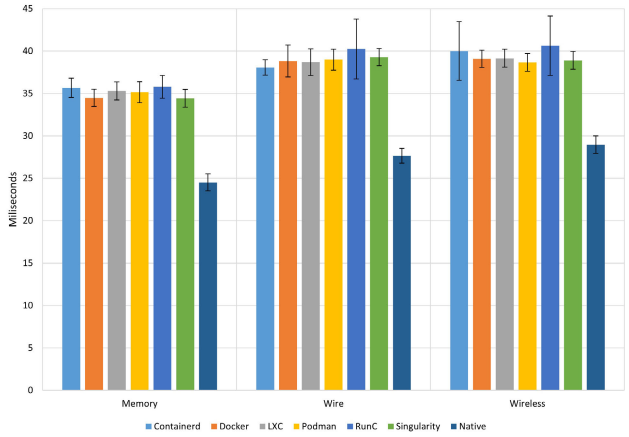


FIGURE 11. Body detection - average processing time.

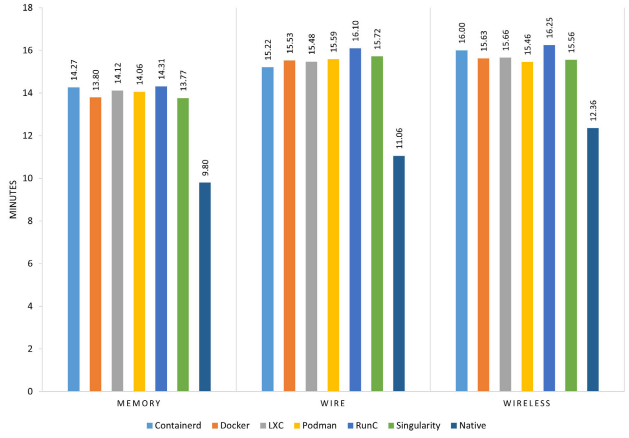


FIGURE 12. Body detection - total processing time.

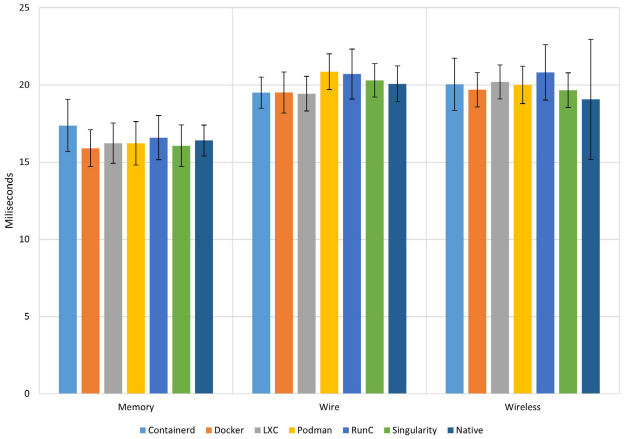


FIGURE 13. Face detection - average processing time.

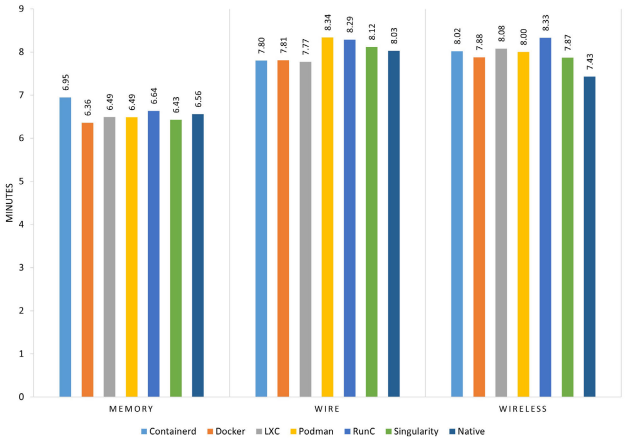


FIGURE 14. Face detection - total processing time.

6.36 minutes. Containerd exhibits the slowest total processing time at 6.95 minutes, while native execution on RPi 4 shows a total processing time of 6.56 minutes.

In the second scenario, LXC leads with a total processing time of 7.77 minutes. Containerd, Docker, Singularity, RunC, and Podman follow closely, with total processing times ranging from 7.80 to 8.34 minutes. Native execution on RPi 4 maintains a total processing time of 8.03 minutes.

In the third scenario, Singularity displays an average processing time of 19.67 milliseconds, resulting in a total processing time of 7.87 minutes, making it the fastest among the container runtimes/engines. Docker follows, with the other containers trailing behind. The native execution on the RPi 4 directly maintains an average processing time of 19.07 milliseconds, resulting in a total processing time of 7.43 minutes.

C. RESOURCE USAGE

This section provides a comprehensive summary of resource usage, highlighting resource utilization that can provide valuable information when operating multiple containers at the same time. The CPU and RAM usage is assessed in the various container runtimes/engines and native execution on the RPi 4 in the three scenarios. CPU usage is measured by the percentage of CPU usage of the four cores of the RPi 4 processor, and the maximum is 400%. In terms of RAM usage, it is in megabytes, noting that the device's RAM is 8 GB.

1) CNN APPLICATIONS

Figure 17 and Figure 18 present the evaluation of CPU and RAM usage for CNN applications across various container runtimes/engines and native execution on the RPi 4 in three distinct scenarios.

In terms of memory usage, the YoloTiny application exhibits a higher memory consumption compared to MobileNet-SSD, with a difference of approximately 30 MB. This variance can be attributed to the neural network configuration within the YoloTiny application and is not related to the choice of container runtimes/engines. Notably, native execution consumes less memory than container runtimes/engines, with a margin of approximately one to six MB in both applications. Moreover, the memory scenario generally utilizes less memory, approximately one MB, compared to other scenarios. However, sporadic increases in average memory usage are observed in certain instances, the cause of which remains undetermined.

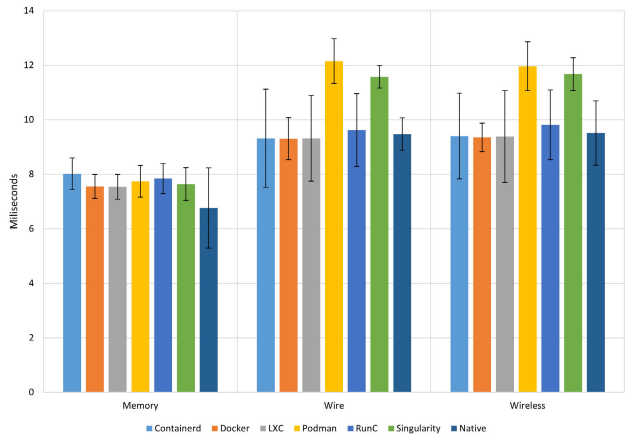


FIGURE 15. Vehicle detection - average processing time.

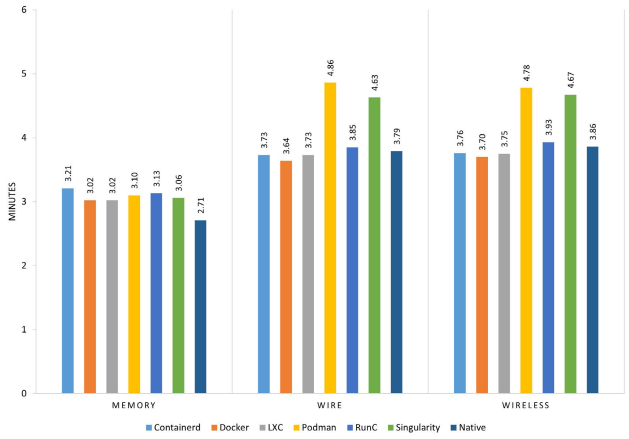
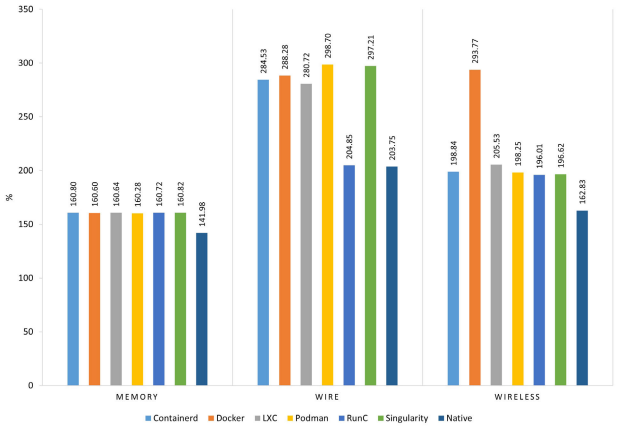


FIGURE 16. Vehicle detection - total processing time.

(a) Object Detection with YoloTiny



(b) Object Detection with MobileNet-SSD

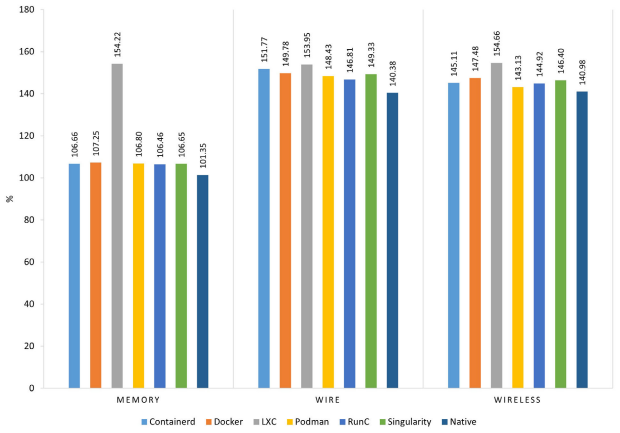


FIGURE 17. Average CPU usage - CNN apps.

Regarding CPU utilization, native execution demonstrates marginally lower CPU usage than container runtimes/engines. Similarly, the memory scenario exhibits slightly lower CPU usage compared to other scenarios. Across all container runtimes/engines, there is uniformity in CPU usage levels. However, notable spikes in CPU utilization are observed in specific cases, such as YoloTiny by Docker in the wireless scenario and MobileNet-SSD by LXC in the memory scenario. Ultimately, both applications in all scenarios, whether in native execution or containerized environments, utilize two out of four cores of the RPi 4 processor.

2) HOG APPLICATION

In the body detection assessment using the HOG algorithm, CPU and RAM usage is evaluated across various container runtimes/engines and native execution on the RPi 4 in three scenarios.

Analyzing the data illustrated in Figure 19 and Figure 20, in the memory scenario, native execution on RPi 4 reveals an average RAM usage of 69.55 MB and an average CPU usage of 79.09%. Among container runtimes/engines, RunC, Containerd, and Docker exhibit average RAM usage ranging from 78.17 MB to 78.52 MB, while others reach the highest average RAM usage at 80.48 MB. Regarding CPU usage, all container runtimes/engines operate in a similar range, spanning from 97.17% to 98.70%.

Moving to the second scenario, native execution maintains an average RAM usage of 77.76 MB and an average CPU usage of 69.65%. Container runtimes/engines, on the other hand, show average RAM usage ranging from 83.44 MB to 85.21 MB. Regarding CPU usage, the average varies between 123.69% for Containerd and 128.40% for Docker.

In the third scenario, native execution on the RPi 4 directly sustains an average RAM usage of 76.98 MB and an average CPU usage of 96.25%. The average RAM usage spans from 81.86 MB for Podman to 83.97 MB for Singularity. Regarding CPU usage, RunC and LXC recorded the lowest CPU usage at 85. 47% and 91. 96%, while others range from 122.37% to 126.32%.

3) HAAR CASCADE APPLICATIONS

Figure 21 and Figure 22 shed light on the assessment of CPU and RAM utilization for Haar Cascade applications across various container runtimes/engines and native execution on the RPi 4, examined within three distinct scenarios.

Regarding memory usage, both face detection and vehicle detection applications demonstrate comparable memory consumption patterns. This similarity arises from the shared

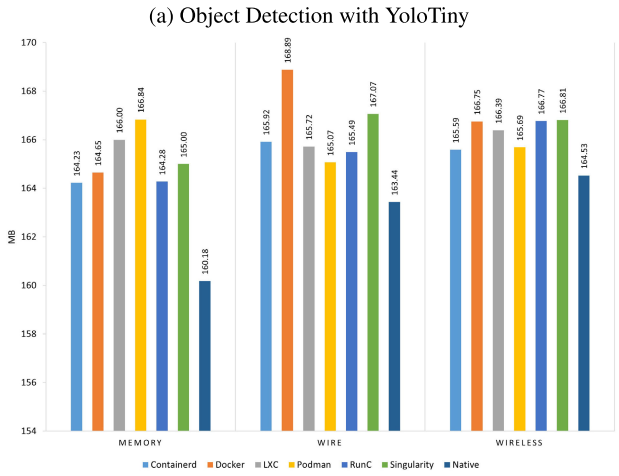


FIGURE 18. Average RAM usage - CNN apps.

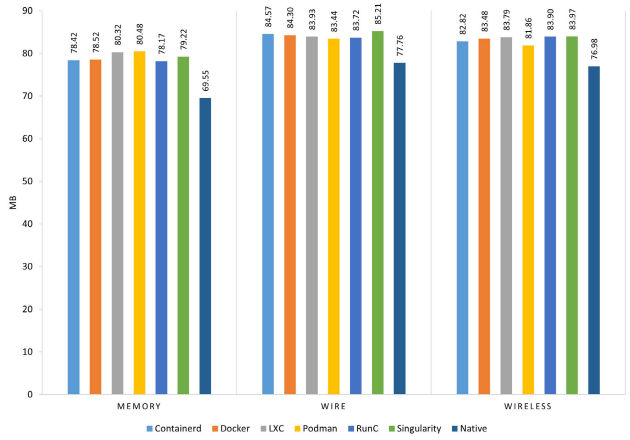


FIGURE 19. Body Detection - Average RAM Usage.

Haar features inherent in both applications, a characteristic that remains consistent across different container runtimes/engines. It is noteworthy that native execution consumes slightly less memory compared to container runtimes/engines, with a variance of approximately 10 to 25 MB across both applications. Additionally, the memory scenario generally exhibits lower memory usage, hovering around one MB, compared to other scenarios. This consistency in memory usage is observed across all container runtimes/engines.

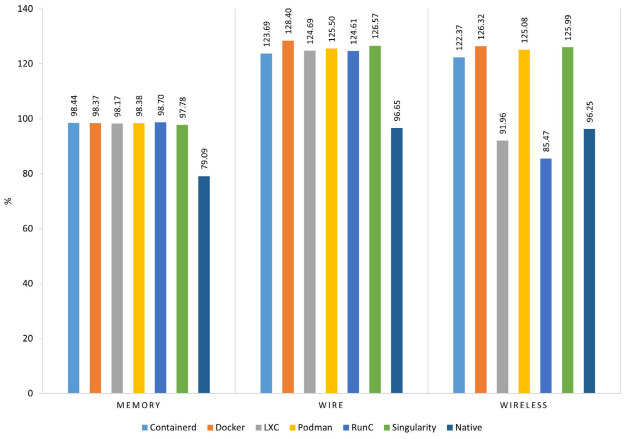


FIGURE 20. Body detection - average CPU usage.

As for CPU utilization, native execution tends to manifest higher CPU usage compared to container runtimes/engines in most scenarios, except for vehicle detection in the memory scenario. Moreover, the memory scenario typically shows marginally lower CPU usage compared to other scenarios in most applications. There is uniformity in CPU usage levels across all container runtimes/engines. However, notable spikes in CPU utilization are noted in specific instances, such as vehicle detection by Docker, Podman, and Singularity in both wire and wireless scenarios. Ultimately, in all scenarios, both applications, whether executed natively or within containerized environments, utilize one of four cores of the RPi 4 processor.

D. TOTAL TIME

This section shows the total time across various computer vision applications is comprehensively analyzed using different container runtimes/engines. The primary objective is to identify the fastest container runtimes/engines for these applications. This analysis is intended to assist in the determination of optimal choices for container runtimes/engines and the evaluation of whether they can be surpassed by native execution in specific scenarios.

Total time covers the entire duration from receiving the first image to processing the last image, including image acquisition, algorithm execution, image saving, and object delineation. This assessment involves the five applications using the six container runtimes/engines, along with direct execution on the RPi 4 in the three scenarios.

1) OBJECT DETECTION BY YOLOTINY

Figure 23 shows that in the first scenario, Docker is the most efficient container runtimes/engines for YOLOTiny object detection, completing the task in 38.27 minutes, closely followed by Singularity with a total processing time of 38.75 minutes and Podman at 38.97 minutes. Subsequently, LXC demonstrates a processing time of 39.91 minutes. Containerd and RunC exhibit slightly longer durations, requiring 40.27 and 40.74 minutes, respectively, to complete

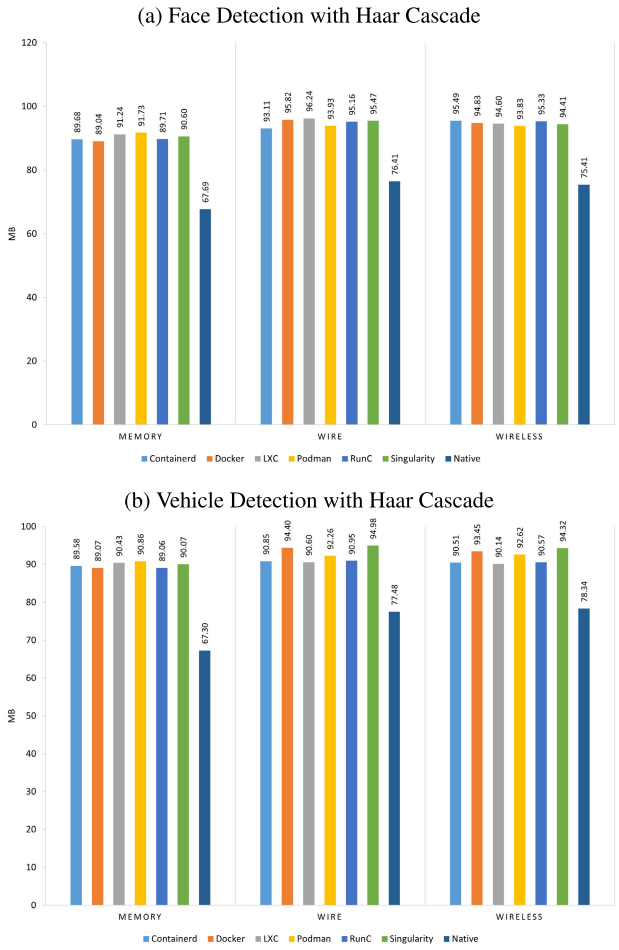


FIGURE 21. Average RAM usage - haar cascade apps.

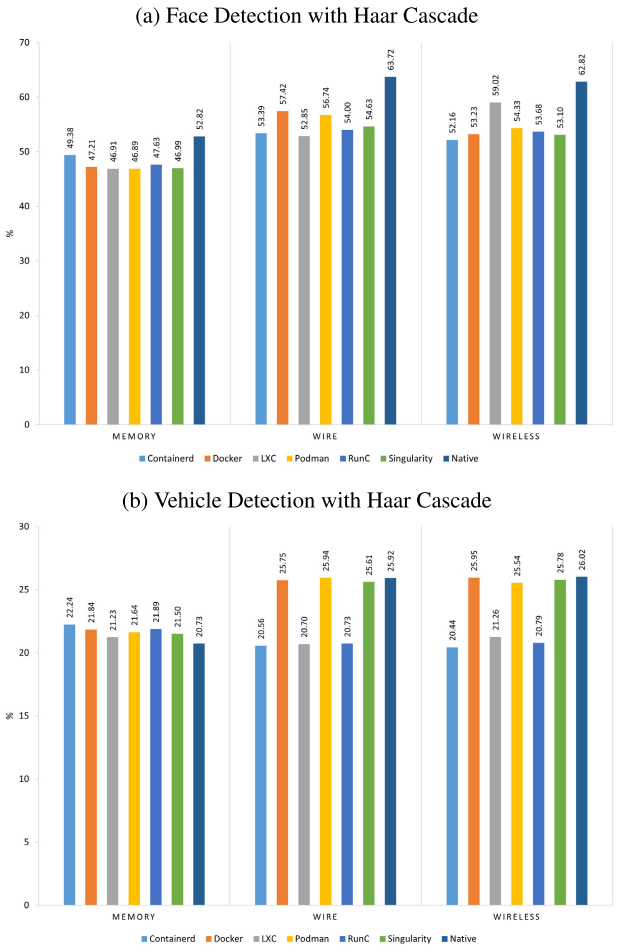


FIGURE 22. Average CPU usage - haar cascade apps.

the task. In particular, native execution on the RPi 4 surpasses the containers, achieving the task in just 35.85 minutes.

In the second scenario, Containerd proves to be the quickest, completing the YOLOTiny application's processing in 41.25 minutes, with Docker closely behind at 41.29 minutes and RunC at 41.66 minutes. Subsequently, LXC shows a processing time of 43.61 minutes, while Singularity and Podman exhibit slightly longer durations at 43.32 and 44.80 minutes, respectively. The native execution on the RPi 4 once again excels, finishing the task in just 39.30 minutes.

In the third scenario, Docker emerges as the most efficient, requiring 42.09 minutes to complete the YOLOTiny application's processing. This is followed by Singularity at 42.57 minutes and LXC with 42.94 minutes. Subsequently, Podman takes 43.49 minutes to complete the entire process. Containerd and RunC display slightly longer durations, with times of 45.16 minutes and 45.48 minutes, respectively. Native execution on the RPi 4 outperforms containers, achieving the task in just 40.39 minutes.

2) OBJECT DETECTION BY MOBILENET-SSD

Figure 24 reveals that in the first scenario, Docker completes the task in 20.71 minutes, closely followed by Singularity at 20.96 minutes. Subsequently, Podman requires 21.56 minutes for the entire process, while RunC takes 21.82 minutes to reach completion. LXC and Containerd exhibit slightly longer durations, with processing times of 22.17 and 22.28 minutes, respectively. Notably, in this scenario, native execution on the RPi 4 surpasses container runtimes/engines, taking only 17.66 minutes to complete the task.respectively.

In the second scenario, Docker remains the fastest, at 21.86 minutes. Singularity follows with 22.11 minutes, and LXC is next with 22.12 minutes to complete the total process. Podman requires 22.25 minutes. However, Containerd and RunC exhibit slightly longer durations, with processing times of 23.69 minutes for both. Native execution on the RPi 4 maintains its superior performance in this scenario, completing the task in just 19.70 minutes.

Moving on to the third scenario, LXC leads at 23.21 minutes, followed by Docker at 23.23 minutes and Singularity at 23.26 minutes. Subsequently, Podman took 23.62 minutes to reach the finish line. Containerd and RunC display slightly longer durations, with processing times of 24.42 and 25.28 minutes, respectively. Once again, native execution on the RPi 4 outperforms container runtimes/engines, requiring only 21.06 minutes to complete the task.

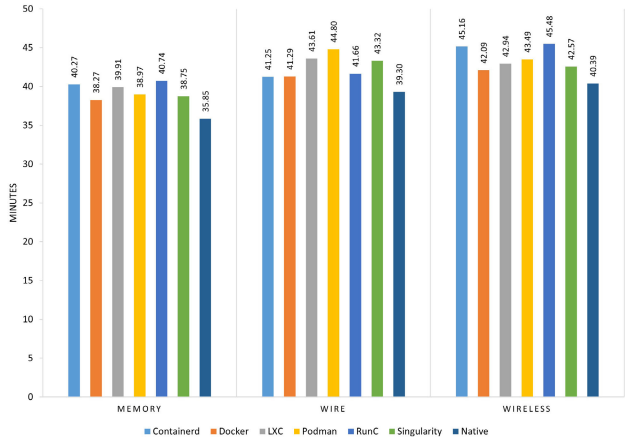


FIGURE 23. YoloTiny - total time.

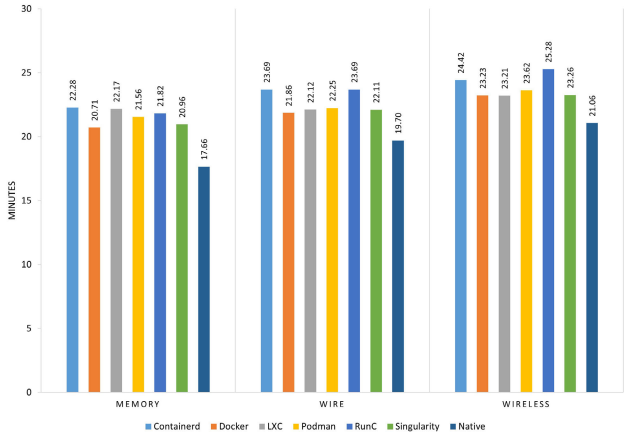


FIGURE 24. MobileNet-SSD - total time.

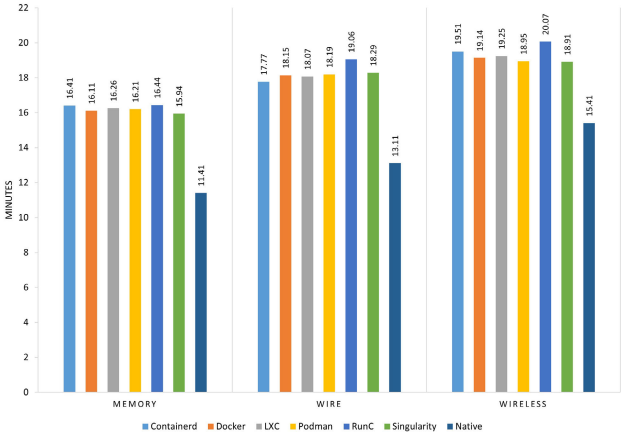


FIGURE 25. Body detection - total time.

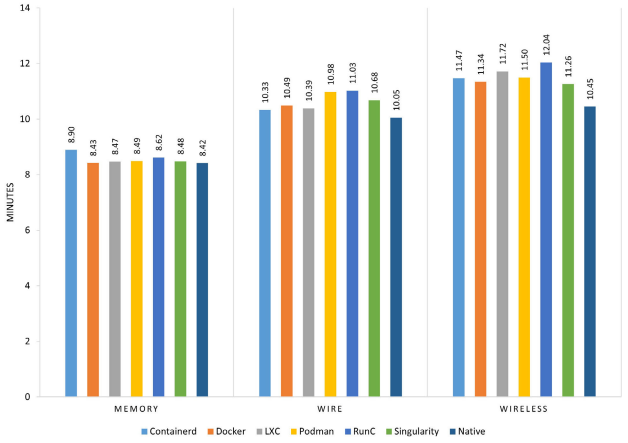


FIGURE 26. Face detection - total time.

3) BODY DETECTION WITH HOG

Figure 25 shows that in the first scenario, Singularity completes the task in 15.94 minutes, with Docker following closely at 16.11 minutes. Subsequently, Podman requires 16.21 minutes for the entire process, while LXC takes 16.26 minutes to complete. Slightly longer durations are observed for Containerd and RunC, with processing times of 16.41 and 16.44 minutes, respectively. In particular, in this scenario, native execution on the RPi 4 outperforms container runtimes/engines, requiring only 11.41 minutes to finalize the task.

Moving on to the second scenario, Containerd completes the task in 17.77 minutes. Following closely, LXC requires 18.07 minutes, while Docker takes 18.15 minutes to complete the total process. Podman follows with a processing time of 18.19 minutes. However, Singularity and RunC exhibit slightly longer durations, with processing times of 18.29 and 19.06 minutes, respectively. In this scenario, once again, native execution on the RPi 4 maintains its superior performance, requiring only 13.11 minutes to complete the task.

In the third scenario, Singularity once more emerges as the swiftest performer, completing the task in 18.91 minutes, followed by Podman at 18.95 minutes and Docker with 19.14 minutes. Then, LXC requires 19.25 minutes to complete the entire process, while Containerd and RunC exhibit slightly longer durations, with processing times of 19.51 and 20.07 minutes, respectively. In this scenario, native execution on the RPi 4 once again outperforms container runtimes/engines, requiring only 15.41 minutes to complete the task.

4) FACE DETECTION WITH HAAR CASCADE

Figure 26 illustrates that in the first scenario, Docker concludes the task in a rapid 8.43 minutes, followed closely by LXC at 8.47 minutes, Singularity at 8.48 minutes and Podman with a similar time frame of 8.49 minutes. Then RunC requires a slightly longer 8.62 minutes to complete the process. Meanwhile, Containerd exhibits a marginally longer processing time of 8.90 minutes. Moving to the second scenario, Containerd achieves the fastest time, completing the task in 10.33 minutes. Following suit is LXC at 10.39 minutes, with Docker requiring 10.49 minutes for task completion. After that, Singularity exhibits a slightly longer duration, with a processing time of 10.68 minutes, while Podman takes a tad longer at 10.98 minutes. RunC, on the other hand, requires 11.03 minutes to conclude the process. Once again, native execution on the RPi 4 asserts its superiority, finishing the task in an impressive 10.05 minutes.

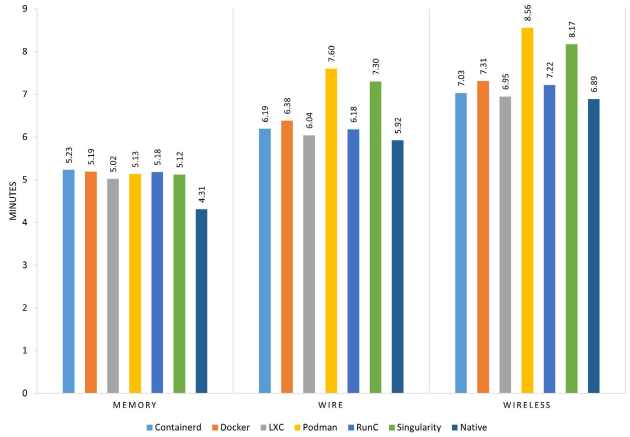


FIGURE 27. Vehicle detection - total time.

In the third scenario, Singularity emerges as the swiftest performer, completing the task in 11.26 minutes, with Docker following closely at 11.34 minutes. Containerd takes 11.47 minutes, and Podman follows suit with a processing time of 11.50 minutes. LXC and RunC exhibit slightly longer durations, requiring 11.72 and 12.04 minutes, respectively, to finalize the process. Remarkably, even in this scenario, native execution on the RPi 4 outperforms the containers, demanding only 10.45 minutes to complete the task.

5) VEHICLE DETECTION WITH HAAR CASCADE

In the first scenario, Figure 27 depicts closely aligned durations among all container runtimes/engines. LXC emerges as the fastest, completing the task in 5.02 minutes. Immediately after, Singularity takes the stage with a brief performance of 5.12 min, closely trailed by Podman at 5.13 min, RunC at 5.18 minutes, and Docker with an almost identical time of 5.19 min. In contrast, Containerd exhibits a slightly longer processing duration of 5.23 minutes. Note that native execution directly on the RPi 4 excels in this scenario, executing the task efficiently in just 4.31 minutes.

Regarding the second scenario, LXC stands out for efficiency, completing the task in 6.04 minutes. Following closely is RunC, which takes 6.18 minutes, and Containerd, requiring 6.19 minutes for task completion. Subsequently, Docker takes 6.38 minutes to complete the process. Singularity endures a slightly longer duration, with a processing time of 7.30 minutes. On the other hand, Podman takes 7.60 minutes to complete the process. Once again, native execution on the RPi 4 asserts its superiority, concluding the task impressively in 5.92 minutes.

In the third scenario, LXC maintains its lead with a task completion time of 6.95 minutes, closely followed by Containerd at 7.03 minutes. RunC takes 7.22 minutes, and Docker takes 7.30 minutes. Singularity and Podman exhibit somewhat longer durations, necessitating 8.17 and 8.56 minutes, respectively, to finalize the process. Notably, even in this scenario, native execution on the RPi 4 outperforms the containers, concluding the task proficiently in just 6.89 minutes.

V. DISCUSSION

This discussion delves into a meticulous analysis of the data derived from our experiments, providing a comprehensive exploration of container technologies optimized for distinct aspects of computer vision applications and overall performance. Upon scrutinizing the data derived from these experiments, a comprehensive summary is presented in Table 3, highlighting the most competent container technologies for aspects of computer vision applications and overall performance. Table 4 shows the top two performers based on the applied evaluation metrics, offering nuanced insights into their comparative effectiveness. Consider that the results and comparisons presented are based on the OpenCV platform for armv7l. Additionally, the Docker container has been adapted to alternative container technologies, potentially influencing their performance. It should be noted that for LXC and RunC container technology, the bridge between containers and the host machine was established manually, possibly impacting image transfer performance.

Containerd excels as an optimal container runtimes/engines to efficiently read images from memory, making it highly suitable for applications that require rapid data retrieval from memory. Regarding network image reception, Containerd, LXC, and Singularity emerge as the top performers. In terms of image processing for each algorithm, Docker stands out as the most effective across all algorithms. Singularity exhibits remarkable similarity in the CNN and HOG algorithms, while LXC demonstrates comparable performance in the Haar Cascade algorithm.

Comparison of resource usage proves challenging due to its inconsistent patterns, with closeness evident in most cases. In specific cases, such as the MobileNet-SSD object detection application and both applications employing the Haar Cascade algorithm, container technology demonstrated reduced resource utilization compared to normal operations. Further experimentation, especially involving multiple containers running together, is recommended to refine our understanding and identify optimal container runtimes/engines for resource usage in various scenarios.

For overall performance in computer vision applications, Docker stands out as the top performer. While it may have a limitation of slower image reads/reception, its faster processing time positions it as the best choice. Specifically, Docker excels in CNN and HOG algorithm applications, sharing the spotlight with Singularity. In applications of Haar Cascade algorithm, Docker proves to be the best, closely followed by LXC.

In terms of resource utilization, the disparity in memory usage across all containers compared to native execution in all three scenarios lacks a clear explanation. Regarding CPU utilization, it was observed that the elevated temperature of the RPi 4 affected CPU usage and contributed to a decrease in performance. This phenomenon may also explain occasional irregular spikes in RAM usage. Further investigation into these aspects is warranted to gain deeper insights.

TABLE 3. Summary of total time, processing time & read/receive time.

Memory Scenario				Wire Scenario				Wireless Scenario			
	Total	Processing	Read/Receive		Total	Processing	Read/Receive		Total	Processing	Read/Receive
Object Detection with YoloTiny											
Native	35.85 min	34.28 min	1.57 min	Native	39.30 min	37.26 min	2.04 min	Native	40.39	37.36 min	3.03 min
Docker	38.27 min	36.27 min	2.00 min	Containerd	41.25 min	39.16 min	2.09 min	Docker	42.09	38.94 min	3.15 min
Singularity	38.75 min	36.83 min	1.92 min	Docker	41.29 min	39.11 min	2.18 min	Singularity	42.57	39.34 min	3.23 min
Podman	38.97 min	37.15 min	1.82 min	RunC	41.66 min	39.43 min	2.23 min	LXC	42.94	39.54 min	3.40 min
LXC	39.91 min	38.13 min	1.78 min	Singularity	43.32 min	41.22 min	2.10 min	Podman	43.49	40.24 min	3.25 min
Containerd	40.27 min	38.54 min	1.73 min	LXC	43.61 min	41.40 min	2.21 min	Containerd	45.16	41.87 min	3.29 min
RunC	40.74 min	38.86 min	1.88 min	Podman	44.80 min	42.65 min	2.15 min	RunC	45.48	41.99 min	3.49 min
Object Detection with MobileNet-SSD											
Native	17.66 min	16.13 min	1.53 min	Native	19.70 min	17.67 min	2.03 min	Native	21.06	18.02 min	3.04 min
Docker	20.71 min	18.42 min	2.30 min	Docker	21.86 min	19.59 min	2.27 min	LXC	23.21	19.90 min	3.31 min
Singularity	20.96 min	18.68 min	2.28 min	Singularity	22.11 min	19.84 min	2.27 min	Docker	23.23	19.85 min	3.38 min
Podman	21.56 min	19.51 min	2.05 min	LXC	22.12 min	19.89 min	2.23 min	Singularity	23.26	20.00 min	3.26 min
RunC	21.82 min	19.79 min	2.04 min	Podman	22.25 min	19.94 min	2.31 min	Podman	23.62	19.93 min	3.69 min
LXC	22.17 min	19.97 min	2.20 min	RunC	23.69 min	21.28 min	2.41 min	Containerd	24.42	21.13 min	3.29 min
Containerd	22.28 min	20.38 min	1.90 min	Containerd	23.69 min	21.50 min	2.19 min	RunC	25.28	21.41 min	3.87 min
Body Detection with HOG											
Native	11.41 min	9.80 min	1.61 min	Native	13.11 min	11.06 min	2.05 min	Native	15.41	12.36 min	3.05 min
Singularity	15.94 min	13.77 min	2.18 min	Containerd	17.77 min	15.22 min	2.55 min	Singularity	18.91	15.56 min	3.35 min
Docker	16.11 min	13.80 min	2.31 min	LXC	18.07 min	15.48 min	2.59 min	Podman	18.95	15.46 min	3.49 min
Podman	16.21 min	14.06 min	2.15 min	Docker	18.15 min	15.53 min	2.62 min	Docker	19.14	15.63 min	3.51 min
LXC	16.26 min	14.12 min	2.14 min	Podman	18.19 min	15.59 min	2.60 min	LXC	19.25	15.66 min	3.59 min
Containerd	16.41 min	14.27 min	2.15 min	Singularity	18.29 min	15.72 min	2.57 min	Containerd	19.51	16.00 min	3.51 min
RunC	16.44 min	14.31 min	2.12 min	RunC	19.06 min	16.10 min	2.96 min	RunC	20.07	16.25 min	3.82 min
Face Detection with Haar Cascade											
Native	8.42 min	6.56 min	1.86 min	Native	10.05 min	8.03 min	2.02 min	Native	10.45	7.43 min	3.02 min
Docker	8.43 min	6.36 min	2.06 min	Containerd	10.33 min	7.80 min	2.53 min	Singularity	11.26	7.87 min	3.39 min
LXC	8.47 min	6.49 min	1.98 min	LXC	10.39 min	7.77 min	2.62 min	Docker	11.34	7.88 min	3.46 min
Singularity	8.48 min	6.43 min	2.06 min	Docker	10.49 min	7.81 min	2.68 min	Containerd	11.47	8.02 min	3.45 min
Podman	8.49 min	6.49 min	2.00 min	Singularity	10.68 min	8.12 min	2.56 min	Podman	11.50	8.00 min	3.50 min
RunC	8.62 min	6.64 min	1.98 min	Podman	10.98 min	8.34 min	2.64 min	LXC	11.72	8.08 min	3.64 min
Containerd	8.90 min	6.95 min	1.95 min	RunC	11.03 min	8.29 min	2.74 min	RunC	12.04	8.33 min	3.71 min
Vehicle Detection with Haar Cascade											
Native	4.31 min	2.71 min	1.60 min	Native	5.92 min	3.79 min	2.13 min	Native	6.89	3.86 min	3.03 min
LXC	5.02 min	3.02 min	2.00 min	LXC	6.04 min	3.73 min	2.31 min	LXC	6.95	3.75 min	3.20 min
Singularity	5.12 min	3.06 min	2.06 min	RunC	6.18 min	3.85 min	2.33 min	Containerd	7.03	3.76 min	3.27 min
Podman	5.13 min	3.10 min	2.03 min	Containerd	6.19 min	3.73 min	2.46 min	RunC	7.22	3.93 min	3.29 min
RunC	5.18 min	3.13 min	2.05 min	Docker	6.38 min	3.64 min	2.74 min	Docker	7.31	3.70 min	3.61 min
Docker	5.19 min	3.02 min	2.17 min	Singularity	7.30 min	4.63 min	2.67 min	Singularity	8.17	4.67 min	3.50 min
Containerd	5.23 min	3.21 min	2.02 min	Podman	7.60 min	4.86 min	2.74 min	Podman	8.56	4.78 min	3.78 min

TABLE 4. Performance summary.

Reading or Receiving Images		
Memory	Wire	Wireless
Containerd -	Containerd LXC	Containerd Singularity
Processing Images		
CNN	HOG	Haar Cascade
Docker Singularity	Docker Singularity	Docker LXC
Total Time		
CNN	HOG	Haar Cascade
Docker Singularity	Docker Singularity	Docker LXC

VI. CONCLUSION AND FUTURE WORK

This paper presents an evaluation of different container technologies, including RunC, LXC, Containerd, Docker, Podman, and Singularity, in the context of computer vision applications on ARM-based edge devices, specifically the RPi 4. The study explores the implementation of computer vision tasks using the OpenCV library, comparing containerized and non-containerized approaches.

Experimental results indicate a discernible performance difference between containerized and non-containerized computer vision applications on edge devices. However, the impact of containerization on performance appears to be moderate. Additionally, the study delves into the performance variations among container runtimes and engines. Containerd exhibits fast memory reading, while Containerd and LXC showcase efficiency in wired image reception, and Singularity and Containerd lead in wireless image reception. Regarding processing time, Docker consistently outperforms other container runtimes/engines across various image processing applications. Despite Docker's potential drawback of slower memory reading, its faster processing time makes it a competitive option.

In conclusion, this study confirms that container technology is a reliable and efficient deployment method for computer vision applications in edge computing. The findings contribute valuable information on the advantages of container platforms in the field of computer vision.

In our current efforts, we have optimized the Docker base image specifically tailored for computer vision applications. Our future work will focus on exploring CPU-based computer vision algorithms across diverse ARM-based edge devices. The aim is to develop and optimize the performance of a heterogeneous edge node cluster dedicated to computer vision applications. This includes managing IoT cameras in real-time scenarios to improve the overall efficiency and effectiveness of edge computing solutions.

REFERENCES

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[2] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[3] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 17, Sep. 2020, Art. no. e5668.

[4] R. Morabito, "Virtualization on Internet of Things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[5] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, Jul. 2019.

[6] E. Cailliau, N. Aerts, L. Noterman, and L. Groote, "A comparative study on containers and related technologies," ResearchGate, Nov. 2016. Accessed: Aug. 16, 2023. [Online]. Available: https://www.researchgate.net/publication/320961475_A_comparative_study_on_containers_and_related_technologies/citations

[7] G. E. de Velp, E. Rivière, and R. Sadre, "Understanding the performance of container execution environments," in *Proc. 6th Int. Workshop Container Technol. Container Clouds*. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 37–42, doi: 10.1145/3429885.3429967.

[8] K. Kaur, T. Dhand, N. Kumar, and S. Zeadally, "Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers," *IEEE Wireless Commun.*, vol. 24, no. 3, pp. 48–56, Jun. 2017.

[9] H. Khazaei, H. Bannazadeh, and A. Leon-Garcia, "SAVI-IoT: A self-managing containerized IoT platform," in *Proc. IEEE 5th Int. Conf. Future Internet Things Cloud (FiCloud)*, Aug. 2017, pp. 227–234.

[10] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, "Exploring container virtualization in IoT clouds," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, May 2016, pp. 1–6.

[11] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May 2018.

[12] R. Morabito, I. Farris, A. Iera, and T. Taleb, "Evaluating performance of containerized IoT services for clustered devices at the network edge," *IEEE Internet Things J.*, vol. 4, no. 4, pp. 1019–1030, Aug. 2017.

[13] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018, doi: 10.1016/j.jss.2018.09.082.

[14] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni, and C. Fetzer, "SGX-aware container orchestration for heterogeneous clusters," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 730–741.

[15] P. Stanojevic, S. Usorac, and N. Stanojev, "Container manager for multiple container runtimes," in *Proc. 44th Int. Conv. Inf., Commun. Electron. Technol. (MIPRO)*, Sep. 2021, pp. 991–994.

[16] O. C. Initiative. *Runc*. GitHub. Accessed: May 16, 2023. [Online]. Available: https://github.com/opencontainers/runc

[17] M. Kirov. (May 2022). *Digging Into Runtimes—Runc*. Accessed: May 16, 2023. [Online]. Available: https://blog.quarkslab.com/digging-into-runtimes-runc.html

[18] L. Containers. *Lxc*. Accessed: Jun. 8, 2021. [Online]. Available: https://linuxcontainers.org

[19] LXC. *LXC*. GitHub. Accessed: Jun. 8, 2021. [Online]. Available: https://github.com/lxc/lxc

[20] Containerd. *Containerd*. Accessed: Jun. 19, 2023. [Online]. Available: https://containerd.io/

[21] Containerd. *Containerd*. GitHub. Accessed: Jun. 19, 2023. [Online]. Available: https://github.com/containerd/containerd

[22] C. Anderson, "Docker [software engineering]," *IEEE Softw.*, vol. 32, no. 3, p. 102, May 2015.

[23] Docker. *Docker Hub*. Accessed: Feb. 23, 2021. [Online]. Available: https://hub.docker.com

[24] O. I. Alqaisi, M. S. Haq, and A. S. Tosun, "Security of containerized computer vision applications," in *Proc. 2nd Int. Conf. Comput. Inf. Technol. (ICCIT)*, Jan. 2022, pp. 115–120.

[25] Sylabs. *Singularity Container Technology & Services*. Accessed: Jul. 26, 2023. [Online]. Available: https://sylabs.io/

[26] Podman. *Podman*. Accessed: Aug. 11, 2022. [Online]. Available: https://podman.io/

[27] Á. Kovács, "Comparison of different Linux containers," in *Proc. 40th Int. Conf. Telecommun. Signal Process. (TSP)*, Jul. 2017, pp. 47–51.

[28] L. Espe, A. Jindal, V. Podolskiy, and M. Gerndt, "Performance evaluation of container runtimes," in *Proc. 10th Int. Conf. Cloud Comput. Services Sci.*, 2020, pp. 273–281.

[29] Z. Li, "Comparison between common virtualization solutions: VMware workstation, hyper-V and Docker," in *Proc. IEEE 3rd Int. Conf. Frontiers Technol. Inf. Comput. (ICFTIC)*, Nov. 2021, pp. 701–707.
[30] Z. Wang, "Can 'micro VM' become the next generation computing platform: Performance comparison between light weight virtual machine, container, and traditional virtual machine," in *Proc. IEEE Int. Conf. Comput. Sci., Artif. Intell. Electron. Eng. (CSAIEE)*, Aug. 2021, pp. 29–34.
[31] W. D. S. Marques, P. S. S. D. Souza, F. D. Rossi, G. D. C. Rodrigues, R. N. Calheiros, M. D. S. Conterato, and T. C. Ferreto, "Evaluating container-based virtualization overhead on the general-purpose IoT platform," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2018, pp. 8–13.
[32] D. Fernández Blanco, F. Le Mouel, T. Lin, and A. Rekik. (Aug. 2023). *Can Software Containerisation Fit the Car On-Board Systems*. Work. Paper or Preprint. [Online]. Available: https://hal.science/hal-04127629
[33] Y. Jing, Z. Qiao, and R. O. Sinnott, "Benchmarking container technologies for IoT environments," in *Proc. 7th Int. Conf. Fog Mobile Edge Comput. (FMEC)*, Dec. 2022, pp. 1–8.
[34] R. Morabito, "A performance evaluation of container technologies on Internet of Things devices," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2016, pp. 999–1000.
[35] A. Acharya, J. Fanguède, M. Paolino, and D. Raho, "A performance benchmarking analysis of hypervisors containers and unikernels on ARMv8 and x86 CPUs," in *Proc. Eur. Conf. Netw. Commun. (EuCNC)*, Jun. 2018, pp. 282–289.
[36] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "KVM, xen and docker: A performance analysis for ARM based NFV and cloud computing," in *Proc. IEEE 3rd Workshop Adv. Inf., Electron. Electr. Eng. (AIEEE)*, Nov. 2015, pp. 1–8.
[37] A. Openwhisk. *Open Source Serverless Cloud Platform*. Accessed: Oct. 6, 2023. [Online]. Available: https://openwhisk.apache.org
[38] Openfaas. *Openfaas-Serverless Functions Made Simple*. Accessed: Oct. 6, 2023. [Online]. Available: https://docs.openfaas.com
[39] vmware Archive. *Kubeless*. Accessed: Oct. 13, 2023. [Online]. Available: https://github.com/vmwarearchive/kubeless
[40] D. Docs. *Swarm Mode Overview*. Accessed: Nov. 21, 2022. [Online]. Available: https://docs.docker.com/engine/swarm/
[41] A. Alabbas, A. Kaushal, O. Almurshed, O. Rana, N. Auluck, and C. Perera, "Performance analysis of apache OpenWhisk across the edge-cloud continuum," in *Proc. IEEE 16th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2023, pp. 401–407.
[42] A. S. Seisa, S. G. Satpute, and G. Nikolakopoulos, "Comparison between Docker and kubernetes based edge architectures for enabling remote model predictive control for aerial robots," in *Proc. IECON 48th Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2022, pp. 1–6.
[43] O. I. Alqaisi, A. S. Tosun, and T. Korkmaz, "Containerized computer vision applications on edge devices," in *Proc. IEEE Int. Conf. Edge Comput. Commun. (EDGE)*, Jul. 2023, pp. 1–11.
[44] R. Pi. *Raspberry Pi 4 Comput. Model B*. Raspberry Pi Trading Ltd. Accessed: Nov. 16, 2022. [Online]. Available: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf
[45] eNetGear. *Nighthawk AC1000 WiFi Router (R6080) User Manual*. NetGear. Accessed: Mar. 28, 2023. [Online]. Available: https://www.downloads.netgear.com/files/GDC/R6080/R6080_UM_EN.pdf?_ga=2.143515121.511659930.1691206616-619111045.1691206616
[46] OpenCV. (Jul. 2012). *OpenCV*. GitHub. [Online]. Available: https://github.com/opencv/opencv/blob/4.x/data/haarcascades/haarcascade_frontalface_default.xml
[47] A. C. Sobral. (Feb. 2016). *Andrewssobral/Vehicle_Detection_Haarcascades*. GitHub. [Online]. Available: https://github.com/andrewssobral/vehicle_detection_haarcascades/blob/master/cars.xml
[48] S. Mallick. (Dec. 2016). *Histogram of Oriented Gradients Explained Using OpenCV*. [Online]. Available: https://learnopencv.com/histogram-of-oriented-gradients/
[49] Z. Yi, S. Yongliang, and Z. Jun, "An improved tiny-YOLOv3 pedestrian detection algorithm," *Optik*, vol. 183, pp. 17–23, Apr. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S003040261930155X
[50] Chuanqi305. *Mobilenet-SSD*. GitHub. Accessed: Sep. 2, 2023. [Online]. Available: https://github.com/chuanqi305/MobileNet-SSD
[51] F. Infomatic. *Vehicle Detection and Counting Using Opencv | Vehicle Counting Using Opencv | Python*. Google Drive. Accessed: Dec. 11, 2021. [Online]. Available: https://drive.google.com/file/d/1QUClOAWRVqNJII1crbCbdStJYzq9Uhfx/view
[52] W. Walker. *Some Cool Motion Sensor Stuff*. Youtube. Accessed: Dec. 16, 2021. [Online]. Available: https://www.youtube.com/watch?v=NyLF8nHIquM&ab_channel=WatchedWalker

OSAMAH I. ALQAISI received the B.S. degree in computer science from Taibah University, Madinah, Saudi Arabia, the M.Sc. degree in computer science from Western Michigan University, Kalamazoo MI, USA. He is currently pursuing the Ph.D. degree in computer science with The University of Texas at San Antonio, San Antonio, TX, USA.

He has gained experience in various roles, including a teaching assistant and a lecturer. He is also a Lecturer with the College of Computing and Information Technology, University of Tabuk, Tabuk, Saudi Arabia. His research interests include containerized computer vision applications, edge computing, the IoT, and computer vision.

ALI ŞAMAN TOSUN received the B.S. degree in computer engineering from Bilkent University, Ankara, Turkey, in 1995, and the M.S. and Ph.D. degrees from The Ohio State University, in 1998 and 2003, respectively.

He was with the Department of Computer Science, The University of Texas at San Antonio, from 2003 to 2021. He is currently the Allen C. Meadors Endowed Chair of computer science with The University of North Carolina at Pembroke, Pembroke, NC, USA. His current research interests include network security, edge computing, software-defined networking, and the Internet of Things.

TURGAY KORKMAZ received the B.Sc. degree (Hons.) in computer science and engineering from Hacettepe University, Ankara, Turkey, in 1994, the first M.Sc. degree in computer engineering from Bilkent University, Ankara, the second M.Sc. degree in computer and information science from Syracuse University, Syracuse, NY, USA, in 1996 and 1997, respectively, and the Ph.D. degree in electrical and computer engineering from The University of Arizona, in December 2001, under the supervision of Dr. M. Krunz.

In January 2002, he joined The University of Texas at San Antonio, as an Assistant Professor with the Computer Science Department, where he is currently a Full Professor. He is also involved in the area of computer networks, networks security, networks measurement and modeling, and internet related technologies. His research interests include quality-of-services (QoS) based networking issues in both wireline and wireless networks.