electronics

MDPI

Review

Containerization in Edge Intelligence: A Review

Lubomir Urblik *, Erik Kajati, Peter Papcun and Iveta Zolotová

Department of Cybernetics and Artificial Intelligence, Faculty of EE & Informatics, Technical University of Kosice, 042 00 Kosice, Slovakia

* Correspondence: lubomir.urblik@tuke.sk

Abstract: The onset of cloud computing brought with it an adoption of containerization—a lightweight form of virtualization, which provides an easy way of developing and deploying solutions across multiple environments and platforms. This paper describes the current use of containers and complementary technologies in software development and the benefits it brings. Certain applications run into obstacles when deployed on the cloud due to the latency it introduces or the amount of data that needs to be processed. These issues are addressed by edge intelligence. This paper describes edge intelligence, the deployment of artificial intelligence close to the data source, the opportunities it brings, along with some examples of practical applications. We also discuss some of the challenges in the development and deployment of edge intelligence solutions and the possible benefits of applying containerization in edge intelligence.

Keywords: containerization; edge intelligence; artificial intelligence; edge computing

check for updates

Citation: Urblik, L.; Kajati, E.; Papcun, P.; Zolotová, I. Containerization in Edge Intelligence: A Review. *Electronics* **2024**, *13*, 1335. https://doi.org/10.3390/electronics13071335

Academic Editor: Palden Lama

Received: 16 February 2024
Revised: 11 March 2024
Accepted: 21 March 2024
Published: 2 April 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

"Data is the new oil" were words Clive Humby spoke in 2006 [1]. Similarly to oil, the value of the data lies in its wide variety of uses, but it is worthless when raw [2]. The recent decade saw a massive rise in cloud computing popularity [3], but this approach has its drawbacks. Berisha et al. [4] mention that the amount of data generated per person was around 1.7 MB per second in 2022, which came out to around 44 zettabytes per day in total. According to a report by ZepDo [5], the global internet traffic reached around 4 zettabytes in 2023. By comparing these values, we can see that a tremendous amount of data never makes its way to the cloud, potentially being wasted without ever extracting any useful information from it.

One of the obstacles in cloud processing is latency, which is caused by the distance between the data source and the cloud. Real-time applications cannot function properly when deployed on the cloud [6]. Edge computing (EC) emerged as a potential solution to some of the problems inherent to cloud computing, mostly latency issues, network dependency, and privacy concerns. We see more companies adopting artificial intelligence (AI) every day. As these models require a large amount of data, they are traditionally trained and deployed in the cloud, where they can access virtually limitless resources. Edge intelligence (EI) is a combination of EC and AI, where the models are deployed on devices located close to the data source, allowing them to either process the data entirely within the edge or in collaboration with the cloud. This approach aims to solve the abovementioned problems and allow for faster, better, and more secure data processing in AI applications.

Modern cloud data centers rely on virtualization to split a single device between multiple users and to prevent a user from accessing another user's data. While more traditional hardware virtualization is still widely used, containerization has come out on top and is the de facto standard in cloud-native applications [7]. Containerization allows us to package our applications together with all the required supplementary files, such as libraries, and deploy them on any supported device, ensuring compatibility and consistency across different environments.

Due to the proliferation of containerization and artificial intelligence in cloud computing, we wanted to take a look at the combination of these technologies in edge intelligence. We will start with a short introduction to edge intelligence and what we consider to be the challenges in developing and deploying such solutions. To understand containerization and other trends in software development, Section 3 will contain descriptions of virtualization, containerization and supplementary technologies and techniques. We will expand on these technologies in Section 4 by describing them in more detail. Section 5 will provide specific examples of edge intelligence solutions utilizing containers in various scenarios. We will then discuss the challenges still present in edge intelligence and how the adoption of containers could address them.

2. Edge Intelligence

The transition from the cloud environment, made up of large and powerful servers, to an edge environment with small devices offering only a fraction of the performance of the cloud alternatives poses a large set of challenges and opportunities [8–10].

The heterogeneity of devices used in edge computing is something not found in cloud computing. The hardware side of the traditional server space is mature and adheres to a set of standards [11]. The computers are made up of standard parts and use common interfaces to communicate. While there is a performance difference between servers, the standardization allows for easier management and better support for different libraries and programs. In edge computing, the devices used can differ massively. The management of such a diverse environment can prove problematic, as the devices use different hardware architectures, different interfaces, and offer different hardware capabilities [12,13].

The performance of edge devices poses a big problem in the field of edge intelligence on all levels. The current approach to artificial intelligence expects a high-performance device to be available, which is not commonly found on the edge [14]. The total performance of edge devices is hindered by multiple factors [15–17]:

- Size—Edge devices often need to be smaller as they need to be placed closer to the data source without being intrusive.
- Power—Edge devices use less power as they do not have the space for a large cooling solution or can be battery-powered.
- Computing performance—Both of the previously mentioned factors result in lower performance of the device as it has to deal with limited resources and space.

More direct access to the data source provides edge devices with unseen opportunities, helps alleviate bandwidth issues, and lowers the reliance on cloud solutions.

3. Containerization

Virtualization allows for the abstraction of the hardware present on a physical machine [18]. This abstraction allows for greater control over access to the devices present on the hardware level of the machine. Modern devices allow for virtualization of various components such as memory, storage, networks, and more. In 2019, the European Edge Computing Consortium presented the Reference Architecture Model Edge Computing [19]. The authors mention virtualization as a vital part of edge computing and, by extension, edge intelligence. The full reference architecture is shown in Figure 1. The virtualization of an entire hardware platform is called hardware virtualization or platform virtualization.

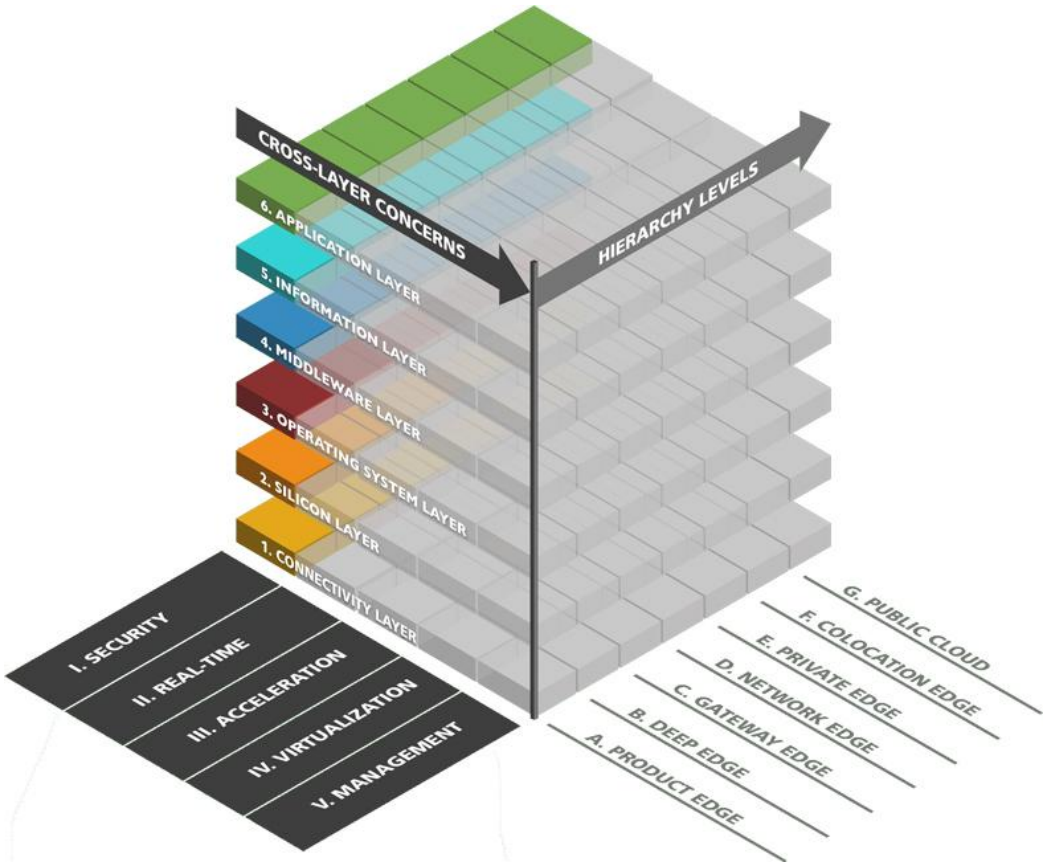

Figure 1. Reference architecture for Model Edge Computing [20].

3.1. Hardware Virtualization

Hardware virtualization separates the operating system from the hardware present on the physical machine. The software or firmware taking care of this separation is called a *hypervisor*. The machine running the hypervisor is called the *host*, and the deployed machines are called *guests*. This virtualization type allows multiple operating systems (OS) to be deployed on the device concurrently. Figure 2 [21,22] shows the two types of hypervisors used.

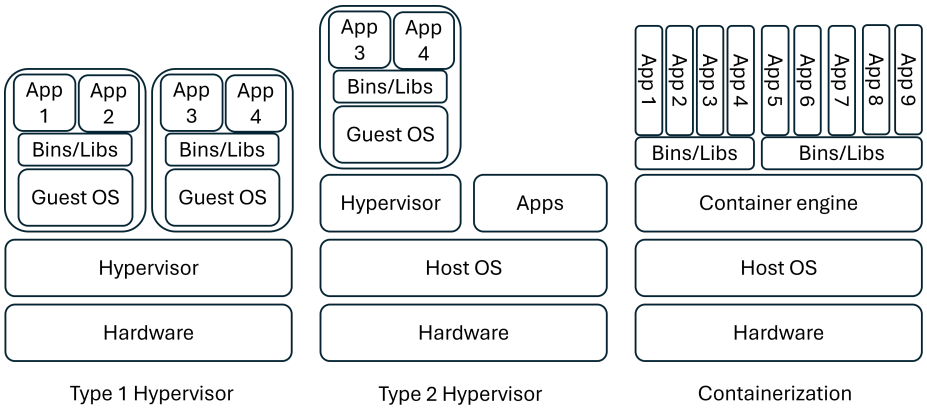

Figure 2. Comparison of hypervisors and containers.

3.1.1. Type 1 Hypervisor

Serving as an additional layer between the hardware and each operating system running on the device, this hypervisor provides drivers for the guest OS to communicate directly with the hardware [23]. The configuration process tends to be more complex when

compared with Type 2 Hypervisors, but the scalability, security, and performance are also higher. Examples of such hypervisors include VMware ESXi, KVM, Oracle VM, and Microsoft Hyper-V.

3.1.2. Type 2 Hypervisor

This type of hypervisor runs on the host OS as any other application would. Unlike a Type 1 Hypervisor, the guest OS communicates with the host OS, which communicates with the hardware. The host is unaware of the guest OS running on the hypervisor and treats it like any other program. The setup required is simpler but also more limited. Examples of such hypervisors include Oracle Virtualbox and VMware Workstation [23]. The traditional use cases include non-production environments where the requirements for performance are not as strict.

3.1.3. Performance Comparison of Type 1 vs. Type 2

There is a measurable difference in performance between Type 1 Hypervisors and Type 2 Hypervisors as described by Dhule et al. [24]. The authors tested five different hypervisors—Type 1 Hypervisors KVM, vSphere, and XenServer, and Type 2 Hypervisors VMWare Workstation and VirtualBox. Each test focused on a different component—CPU performance, memory performance, disk I/O performance, and network performance. The biggest difference they measured was in CPU performance, where VirtualBox reached a little below 60% of native performance and VMWare Workstation slightly above. The lowest among Type 1 Hypervisors was XenServer, with 79%. KVM and vSphere reached around 87% and 94%, respectively. The performance differences were much smaller in memory tests, with VirtualBox and VMWare Workstation achieving around 67% and 75%, respectively. Type 1 Hypervisors all achieved between 90 and 94%, much closer to the native performance. In disk I/O tests, KVM was the clear winner, achieving around 98–99% of native performance. vSphere achieved around 94% and XenServer around 90%. The performance of Type 2 Hypervisors switched during this testing, with VirtualBox coming ahead of VMWare Workstation at around 77% to 73%. The last testing performed focused on networking. KVM and vSphere achieved similar performance at around 90%, with XenServer slightly behind at around 87%. VirtualBox achieved around 61% and VMWare Workstation around 68%. The differences between Type 1 and Type 2 Hypervisors vary based on the area of focus, but Type 1 achieved better performance in all scenarios.

3.2. Containerization—Operating System Level Virtualization

In recent years, the rise of OS-level virtualization, also known as containerization, can be attributed to the shift in software solutions architecture. Traditionally, software solutions were created as a singular block, also known as a *monolithic architecture*, in which different parts of the solution were interconnected inside the application itself. This approach can be easier to develop, test, and maintain in smaller solutions, but as the complexity of the solution increases, so does the maintenance required to keep it working. In the era of cloud computing, a new architecture was adopted—a *microservice architecture*. In microservice architecture, different parts of the solution are split into separate applications interconnected through a common interface. This separation allows for the solution's greater customizability, reusability, and scalability. Separate services can be modified and deployed without affecting any of the other services. They can also be scaled and deployed independently, according to the user needs [25]. The current approaches to software development are closely connected with containerization. The application of DevOps methodology is intertwined with CI/CD, which utilizes containers during the product lifecycle.

3.2.1. Definition

In hardware virtualization, each guest runs in a separate virtual machine with its own kernel. Running multiple versions of the same kernel or different kernels altogether is not

a problem. In OS-level virtualization, the host kernel is shared among all the containers. Running different kernels, for example, the Linux kernel and the Windows kernel, is currently not possible. This distinction is perhaps the biggest strength while also the biggest weakness of containers. As shown in Figure 2, the container engine runs on top of the host OS [26].

Isolation is integral to virtualization as it provides additional security and prevents mishaps from damaging other guests. In hardware virtualization, the isolation is managed by a hypervisor, and each guest has a separate kernel. The kernel is shared in containers, and different techniques must be employed to achieve similar results. Containers take advantage of the features of the Linux kernel, more specifically *namespaces* and *cgroups*.

3.2.2. Namespaces

Namespaces divide the kernel resources between processes, so a set of resources is only visible to selected processes [27]. By utilizing namespaces, each container can see a different set of resources such as host name, process identification number, mount points, networks, inter-process communication, and even time. Unless specified otherwise, the containers are unaware of other containers running the host using different namespaces. They cannot see other processes, networks, or data stored on the host [28].

3.2.3. Cgroups

Cgroups is a "Resource management and resource tracking" solution in the Linux kernel [29]. In hardware virtualization, the hypervisor is told what resources to allocate to which guest [30], be it CPU cores, RAM, or even storage. The kernel is shared in containers, as are the available resources. To prevent a single container from consuming all resources due to the high hardware requirements or a bug in the code, cgroups can divide the available hardware resources between containers [31].

3.3. Performance of OS-Level Virtualization

OS-level virtualization tends to require fewer resources than hardware virtualization. It is considered more lightweight because it does not entail the performance overhead of virtual machines. However, the performance difference is not as clear-cut as it might seem.

Aniruddh et al. [32] compared the performance of VMs and containers in multiple scenarios representing common uses. In the first test, which represented Infrastructure as a Service (IaaS), the performance difference was minimal, with hardware virtualization performing slightly better in more tests. The test compared the CPU and RAM usage during a sysbench test. In the second test, which represented Platform as a Service (PaaS), the container was a clear winner, outperforming the VM by a large margin. MySQL was used as a database, and the authors compared the speed and the hardware utilization during read–write–access operations. In the final test, which represented Software as a Service (SaaS) , the container was again a winner with almost a 100% better performance in some scenarios. The scenario contained two parts—a MySQL database and a Django website, which performed CRUD (Create, Read, Update, Delete) operations on the database.

Abuabdo and Al-Sharif [33] focused on testing multi-threaded algorithms comprising matrix multiplication written in both Java and C. These algorithms were then tested on two virtual machines—Windows 10 and Ubuntu—and in a Docker container. Both the Ubuntu machine and the container performed similarly to the host machine in single-threaded tests with smaller matrices but performed measurably worse with larger matrices. Only the Windows 10 VM could utilize the multi-threaded performance, but only in some scenarios and to a much lesser degree than the native OS.

Watada et al. [34] provide an excellent overview and comparison of hypervisors and containers in various tasks. The authors tested random access performance, CPU performance, network, and memory bandwidth performance. Their tests show that containers achieve performance similar to or better than other virtualization techniques in some tasks while lagging behind in others.

3.4. Containers and Data

Containers are ephemeral—the data in them is stored only for a short time and can be lost at any time. It is not uncommon to re-deploy containers every few days. Each deployment creates a new container with its own data, separated from other containers or the host. The data should, therefore, be separated from the processing itself to prevent losses.

Containers stop whenever they finish their tasks. Because of this, there is a high risk of losing any data not stored elsewhere. The easiest way to share data between containers and the host is using *volumes*. Volumes create a shared folder that persists through container deletion and can be shared to multiple containers.

As edge intelligence mostly deals with real-time data, the data must somehow enter the container. This is often achieved using an API inside the container, which allows the user to send any data into the container for processing. Data can be exchanged between containers through internal container networking, which Docker supports. These networks are separated from the host and each other.

3.5. DevOps

DevOps combines two important parts of a product lifecycle—development and operations. It is described as a set of principles or methods bridging the gap between the development and operations teams [35]. It is not a set of tools but rather guidelines or best practices to which the teams should adhere. Consisting of eight steps, shown in Figure 3, the DevOps cycle is a collaboration and a seamless transition from operations to development and vice versa. By bridging them together, the product can be delivered faster, more reliably, and to the specification of the consumer [36]. The team receives feature requests from the customers, which are translated into a set of tasks for the developers. The tasks are planned ahead of development to ensure the feasibility of delivering them on time. The developers then write the necessary code, which in turn is built and tested to catch errors and bugs as soon as possible. After the feature is finished, the application is ready to be released and awaits deployment. The application is then deployed and released to the customers, where it is monitored using various performance tools that collect data. The new feedback is collected along with the data, a new set of feature requests is collected from the customers, and the cycle repeats.

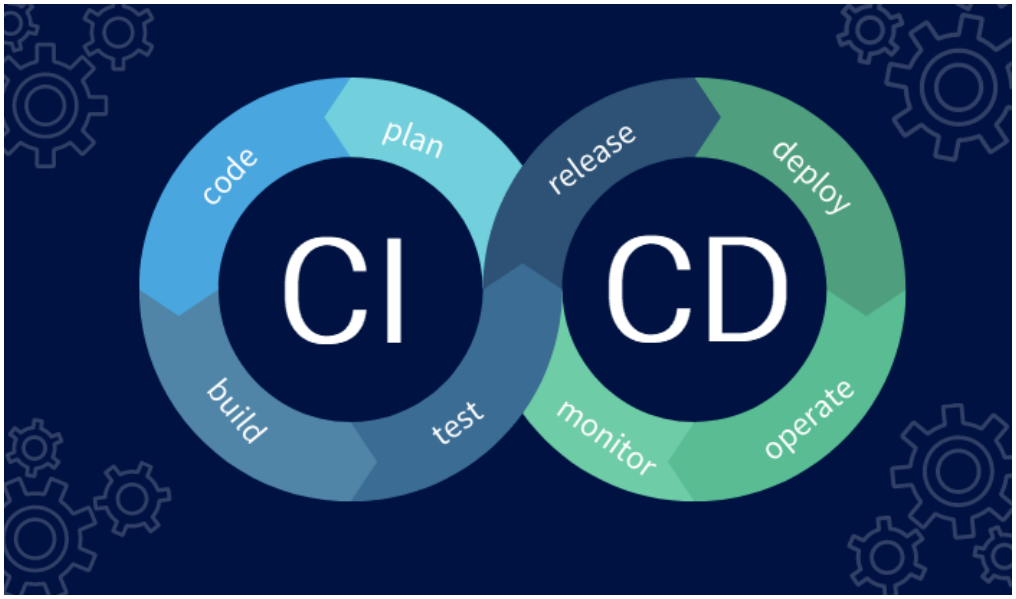

Figure 3. CI/CD pipeline diagram.

3.6. CI/CD

Continuous integration and continuous delivery (CI/CD) has become essential to developing and maintaining software and is vital to the DevOps cycle [37]. The current

agile approach in software development leads to small changes being made to the product. These changes then need to be deployed in small increments, where either the whole application or a part of it needs to be rebuilt and re-deployed. The idea of introducing automation into the development, testing, deployment, and monitoring leads to a shorter lifecycle of apps [38].

In continuous integration, developers upload their changes more often, and the application is tested using automated testing software before being integrated into the code base. This type of automation can find bugs quickly and without any user interaction. Conflicts can occur in a multi-developer environment where each developer has their own tasks. Fixing such conflicts is time-consuming and often needs to be performed manually. By uploading the changes in smaller increments and testing often, the conflicts can be detected early and fixed before they become a bigger problem.

Continuous delivery is the next stage, where the code is prepared to be deployed to any environment at any time. The implementation of the CI/CD is referred to as a *CI/CD pipeline*, where the stages take place in succession after the previous one has successfully finished. An example of such a pipeline is shown in Figure 3. According to a survey by JetBrains from 2023, [39], 63% of developers use Docker, the most used container software , during the CI/CD lifecycle.

Containerization allows the application to be packaged with the required libraries, supplementary software, and files. This package is also called an *image*. The image can then be shared and deployed on another device where it is deployed as a *container*. The container runtime should ensure that the image behaves consistently across different devices. The process of packaging, uploading, and deploying can be fully automated inside the CI/CD pipeline, leading to a shorter time from writing code to deploying it.

3.7. Security

Containers are flexible and used in a wide variety of ways. The recommended way to use containers is as part of the microservice architecture, where each container runs a single service and contains only the tools necessary to perform this service. Adhering to this approach reduces the vulnerabilities by reducing the number of possible attack vectors. The problem comes when the size of the image used increases. Including more libraries and background processes in the container increases the possibility of a vulnerability in any one of them [40,41].

Using containers as a replacement for a VM has also become widespread use. As with a single application, the entire OS can be packaged together into a single image and shipped to users. This increases hardware requirements, as the container now runs multiple processes and services, opens more communication channels between the container and the host, and provides a much larger attack surface [42]. As previously mentioned, containers share a single kernel between themselves and the host. In hardware virtualization, the guests are separated from the host, and the hypervisor is responsible for the security. By attacking the hypervisor, each guest can then be attacked. Vulnerabilities in the OS running in a guest VM can also be an attack vector, allowing the guest to start communicating directly with the host, bypassing the hypervisor [43].

3.8. Creating a Container Image

Containerizing an application requires the creation of an image—a blueprint for the containers. The Open Container Initiative (OCI) [44] provides definitions and specifications of an OCI-compliant image, but it is not necessary for a user to be familiar with said specifications. We will use Docker images as an example, as they are OCI-compliant and represent the widest user base. The images comprise two main parts—the application and the configuration.

The *Dockerfile* serves as the configuration file, containing instructions on how to build and run the image. An important feature of containers is the ability to build images from

other images. An example Dockerfile is shown in Listing 1, containing the instructions to create a Flask application.

Listing 1. Dockerfile Example.

```
FROM python:3.8-alpine
WORKDIR /app
RUN pip install flask
COPY ./app.py /app
ENV FLASK_APP app.py
ENTRYPOINT [ "flask" ]
CMD ["run", "--host", "0.0.0.0"]
```

The first command, *FROM*, defines the base image—in this case, Python, version 3.8, with Alpine serving as the environment. *WORKDIR* configures the working directory of the image. *RUN* executes the commands while building the image. This step is typically used to download additional libraries and tools that the application needs. Using this command when creating an image ensures a constant environment, preventing dependency incompatibility stemming from software updates. *COPY* is used to copy files from our system to the container. These files can be the source code, compiled application, or supplementary files. *ENV* configures the environment variables and should be used to define values which should not change during deployment or require a default value. The last two commands, *ENTRYPOINT* and *CMD*, define the instructions executed when running a container. Each command creates a new layer, a read-only set of files. The layers are stacked on each other, creating an image. Any change to the image requires the creation of a new layer. More commands are available, as documented by Docker [45], but the abovementioned commands represent a minimal setup required to containerize an application.

After creating the image, it can be deployed on any device supporting the selected platform or other compatible platforms.

4. Tools

A wide range of tools is available for containers, spanning from container runtimes to orchestrators. This section will describe the most popular offerings in different categories.

4.1. Docker

Docker is perhaps the most well-known container platform as it offers an easy-to-understand set of tools to package, deploy, and run applications across different platforms and devices [46]. It is an open-source platform and an establishing member of the OCI, which aims to establish a set of open standards for containers. The name "Docker" has been used to refer to almost any part of the Docker platform, from the highest level tools offered—the Docker Desktop and the Docker CLI, both of which provide the user with an easy way of sending commands—to a lower-level tool, the Docker daemon, which will be described later. The use of Docker as an edge computing platform has been described by Ismail et al. [47].

Docker client refers to the user interface, which can be graphical or terminal. Docker offers tens of commands to build, manage, deploy, and monitor containers. The commands can be modified using hundreds of options and arguments, giving the user complete control over the containers. An example is setting environmental variables inside the container or limiting the memory available to the container. The client then relays the commands to the server, the Docker daemon.

Docker daemon is a service running on the host computer and takes on a role similar to a hypervisor in hardware virtualization. The daemon executes the requested command whenever the user enters a command using the client. It is also responsible for the management of other *objects*, which include images, containers, networks, volumes, plugins, and more. Whenever a user requests an image that is unavailable locally, the daemon

communicates with the *image registry*, an external storage for container images. It ensures that the image is downloaded and added to the local registry. The daemon can then take the image, which serves as a template, and create a container according to the specifications of the image. The division of containers into namespaces and cgroups also falls under the scope of the daemon's responsibilities, providing the needed process, storage and network isolation.

The creation and management of containers themselves is delegated to another daemon referred to as a *low-level container runtime*, which in Docker is *containerd*. The user does not interact with the runtime; the entire responsibility is on the Docker daemon instead. *Containerd* edges the line between a high-level container runtime, with which the user can communicate via an API, and a low-level runtime, with which the user has no interaction. In practice, *containerd* is rarely used by itself and is instead part of a more extensive system where another service takes the place of the high-level container runtime.

One of the most significant drawbacks of the Docker platform was and still is the need to run the containers with root privileges, sparking security concerns. This issue is well known to the creators [48] as well as researchers [41,42]. Some solutions were developed [49,50]; however, the problem has created other tools, such as Podman, Buildah, runC, Buildkit, LXD, and *Containerd*, which aim to fix this and other vulnerabilities and prevent malicious attacks through containers.

4.2. Kubernetes

Docker is a great tool when running a small set of applications on several devices. It is not a great choice when dealing with a large number of applications or running an app on many devices. Container orchestrators are tools to automate some parts of the container lifecycle, namely, the provisioning of resources, deployment, scaling, networking, and self-healing [51]. Kubernetes (k8s) is the most used container orchestrator, with reports hovering around 70% market share [52].

Kubernetes (k8s) operates based on *Pods*, which package single or multiple containers with volumes running in the same environment [53]. They usually represent a single application or a part of an application, and all the containers in a Pod run on the same device, which k8s calls *nodes*. When deploying new Pods, another object is created first—*Deployment*. A Deployment contains the desired state of our Pods, which entails the definitions of Pods and a number of replicas to deploy. The Deployment then creates another object called *ReplicaSet*, whose purpose is to scale the Pods according to specification and maintain them in a healthy state. Deployments and ReplicaSets sound similar but differ in their responsibilities. Deployments are responsible for the entire state of the deployed solution and ensure updates and rollbacks. ReplicaSets are responsible for achieving the desired state defined in a Deployment and are not recommended to be used directly outside of Deployments.

Many different k8s distributions are aimed for use in EC, such as k3s, k0s, or Microk8s. Edge devices are often performance-constrained, and the low-performance overhead that comes with containerization while maintaining all the other advantages—such as easy deployment, consistency, and portability—can help create edge solutions. The devices also often support different operating systems or different versions of the same operating system, which leads to compatibility issues, which the use of containers can also address. Using ReplicaSets, the solution can also be easily distributed across multiple devices, allowing for better load balancing and better reliability [54].

4.3. Anaconda

Anaconda, or more specifically, its package manager, *conda*, is a popular choice for the development of AI, ML, or data science applications [55]. There are some areas between containers and conda that overlap, so we see it appropriate to mention it here.

Conda is a package manager [56] responsible for installing and managing Python packages and libraries. It was created as an alternative to *pip*, the default package manager

used in Python. Pip installed the requested package and all the required dependencies without checking for any conflicts with previously installed packages. This could result in broken environments or incompatible versions of packages, as it only cared about the newest package and its dependencies. Conda, on the other hand, checks for conflicts across all the installed packages and tries to solve any conflicts that arise. These packages are installed inside separate environments, allowing multiple versions of the same package to be used on the same device. The developer can switch between these environments easily and ensure the correct versions are used for each project.

The environments can be shared similarly to containers, but these platforms have some major differences. Conda is responsible for the packages, or libraries, used in the solution and is limited to Python and R. Docker, or similar containerization platforms, allow for the management of the entire environment. This includes OS libraries, other software components, or files. The biggest difference is in the execution of the software. Docker is responsible for the execution of the container and the separation from other processes running on the device. Conda does not execute the software and only prepares the environment for Python to execute the program correctly. It is more of an environment manager and not a virtualization platform.

5. Solutions and Architectures on Edge

Three hierarchical levels are typically used to divide up the computer resources. The hierarchy is depicted in the context of IoT in Figure 4; however, this divide is not exclusive and may also be used in other paradigms. The data sources—things, in the case of the Internet of Things—are positioned at the bottom of the hierarchy since they are the most numerous and have the least amount of processing capacity available. Depending on the topic of interest, several data sources can be used instead of these.

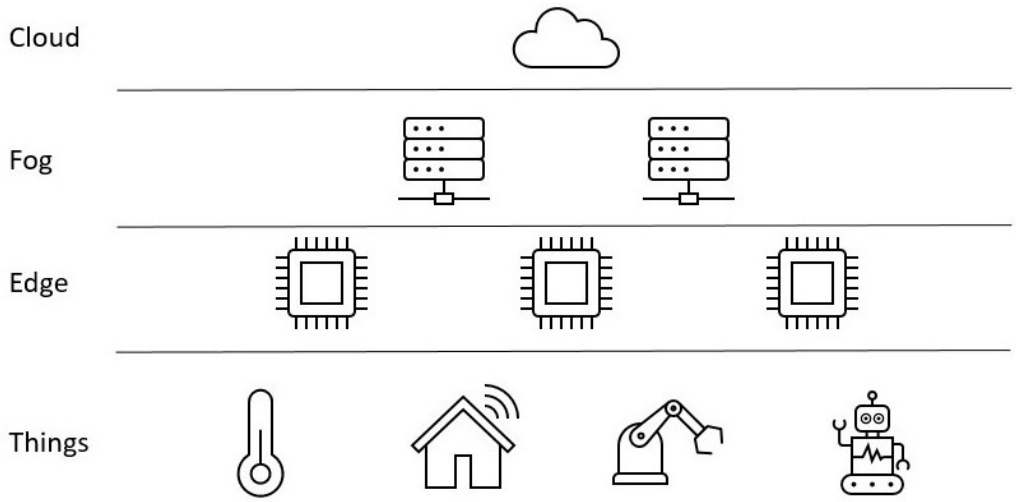

Figure 4. Traditional three-layered computing architecture, made up of edge, fog, and cloud.

The cloud serves as the final destination for most of the data. The cloud is defined by NIST [57] by the following properties:

- On-demand self-service—The user can automatically provision resources and services through an online interface.
- Broad network access—Access to the resources and services is available through standard networks and interfaces, such as the Internet.
- Resource pooling—The provider's resources are pooled to serve multiple consumers, with dynamic assignment and reassignment of virtual and physical resources according to consumer demand.
- Rapid elasticity—The resources available to the user should be scalable both up and down without much user configuration. The process should be available at any time.

- Measured service—The usage is monitored according to appropriate criteria (storage, processing time, bandwidth, etc.), and these metrics are available to the user at any time, together with the associated costs.

The cloud is a collection of (almost) limitless resources accessible via the Internet at any time or location. One benefit of cloud computing is that part of the duties is delegated to the provider, such as network management, cooling and cleaning, hardware upgrades, and updates. The primary drawback of cloud computing is the delay in communications. There is a notable latency due to the distance and the number of network hops needed to get to the cloud, as the application's data centers may be hundreds of kilometers away. Because of this, real-time applications are not appropriate for the cloud.

The fog serves as an extension of the cloud closer to the network's edge. According to the OpenFog Consortium [58], the fog should preserve all the benefits of the cloud while offering other advantages. The advantages mentioned by the OpenFog Consortium are:

- Security—Additional security to ensure safe, trusted transactions.
- Cognition—Awareness of client-centric objectives to enable autonomy.
- Agility—Rapid innovation and affordable scaling under a common infrastructure.
- Latency—Real-time processing and cyber-physical system control.
- Efficiency—Dynamic pooling of local unused resources from participating end-user devices.

The computer resources that are near the data source are represented as the edge. The edge's maximum distance from the data source is not well defined. Most people agree that the edge must have real-time data processing capabilities; therefore, to avoid latency problems, the processing should happen locally on the network. The processing may happen on the same device gathering the data or on a different nearby device. Depending on the use case, these devices can be MCUs, SBCs, Mini-PCs, or mini datacenters. The latency, size, and performance requirements for edge computing vary depending on the use case.

The differences between fog and edge are not clearly defined and are a subject of debate amongst both researchers and industry. We understand fog as the layer between the edge and the cloud, which moves some of the tasks from the cloud closer to the edge while not offering the full advantages of the edge. We see fog as more fit for short-term data storage and data analysis where some processing takes place, whereas edge is for real-time data processing. The definition of how long the real-time data processing should last varies across different scenarios. However, we understand it as the timeframe during which the value of the data does not diminish and the reaction to the data results in a positive change.

Zhou et al. [59] divide edge intelligence into six levels, which are summarized in Table 1 and described in more detail later:

1. Cloud-Edge Coinference and Cloud Training—the model is trained in the cloud, where the performance allows for faster and better performance of the model. The data from the edge is partially offloaded to the cloud for inference.
2. In-Edge Coinference and Cloud training—the model is trained in the cloud, as in the previous level. The inference occurs in-edge, where the data can be fully or partially offloaded to nearby nodes or devices without leaving the edge.
3. On-Device Inference and Cloud Training—the model is trained in the cloud, but the inference occurs on the edge device itself. No data offloading is conducted.
4. Cloud-Edge Cotraining and Inference—both the model training and inference are conducted in cooperation between the edge and the cloud.
5. All In-Edge—Both the training and the inference are conducted in-edge, with edge devices cooperating and offloading data.
6. All On-Device—Both the training and the inference are conducted on-device, with each device working by itself.

The authors [59] focused on deep learning, the most promising part of artificial intelligence, which has seen massive leaps in the last few years. In 2023, OpenAI released

ChatGPT 4, a deep-learning large language model that sparked a widespread interest in artificial intelligence. According to an article from Bilan [60], 49% of the top 1000 companies utilize ChatGPT, and 30% more plan to in the future. The main areas of adoption are marketing, customer support, and programming. The training of such models is very hardware-intensive and is, therefore, typically performed on the cloud. The main disadvantage of a cloud-based approach is the need for the data to travel to the cloud. Devices are producing much more data than the cloud can transfer and process in a reasonable time. This has sparked an interest in edge intelligence, an approach to artificial intelligence taking advantage of the environment around the data sources. Moving some or all of the processing closer to the data can increase the total performance and decrease the processing time.

Table 1. Description of training and inference on edge intelligence levels.

Level	Training	Training Description	Inference	Inference Description
1	Cloud	Data are aggregated in the cloud, where the entire model is trained	Cloud-Edge	The model is split between the edge and the cloud, utilizing methods such as network splitting
2	Cloud	Data are aggregated in the cloud where the entire model is trained	In-Edge	The model can be split between multiple edge devices like network splitting, or use dedicated inference nodes to which the data are offloaded
3	Cloud	Data are aggregated in the cloud, where the entire model is trained	On-Device	The cloud-trained model is deployed on a single device, often utilizing methods such as pruning, quantization, or knowledge distillation to achieve better performance
4	Cloud-Edge	Data are shared between the cloud and the edge; both parts are responsible for a part of the training, utilizing methods such as network splitting	Cloud-Edge	The model is split between the edge and the cloud, utilizing methods such as network splitting
5	In-Edge	Data are shared between edge devices with part of the training taking place at each device or a subset of dedicated training devices	In-Edge	The model can be split between multiple edge devices in a manner similar to network splitting or use dedicated inference nodes to which the data is offloaded
6	On-Device	Data are not shared anywhere, and a single device is responsible for the entire training, which is often optimized for that specific device	On-Device	The model is deployed on a single device, often utilizing approaches such as TinyML

In Level 1 edge intelligence, the model is trained strictly in the cloud, as it offers incomparable performance to the edge. During inference, the edge and the cloud cooperate by exchanging the data. There are multiple approaches to this cooperation. *Network splitting* separates the model into two parts—one for edge deployment and one for cloud deployment [61]. The NN can be split into layers, with the first layers of the network deployed on the edge and the later layers deployed in the cloud. Applying some of the processing on the edge lowers the amount of data sent to the cloud. The bandwidth required is, therefore, lower and could lead to improved latency. Banitalebi-Dehkordi et al. [62] applied this approach and expanded on it by trying to find the ideal split. The model needs to be split correctly to ensure the balance between the network latency and edge performance.

In Level 2, the training is the same as in Level 1. The difference lies in the inference. The nodes can share the data they are to process with other nodes on the edge. This offloading can be performed fully, where the device offloads the entire processing to another node. In partial offloading, the approach is similar to the node splitting mentioned above, where part of the processing takes place on one node and the rest on another node. By removing the need for a constant cloud connection, the external network bandwidth requirements are

lower. On the other hand, the requirements for the local network increase to ensure smooth data sharing. The node performance requirements also rise, as the processing should be conducted in real-time. Yang et al. [63] proposed a pipelined cooperation scheme in which the NN is split into layer segments and assigned to device clusters. The authors then compared their approach with three other parallelization approaches, with the proposed approach being 1.7–6.5 times faster.

Level 3 edge intelligence relies on the cloud for training and a single device for inference. As previously mentioned, AI models tend to require more performance than what typical edge devices can offer. To address this issue, multiple techniques were developed to transfer larger models to lower power devices. The first such approach is *pruning*—a process of removing elements from a model [64]. The complexity of neural networks introduces unnecessary elements that can be removed with little to no effect on the precision of the network. The removed elements include neurons, weights, or entire layers. The second approach is *quantization*—the reduction in elements' precision representation [65]. YOLO, a popular real-time object detection system, has undergone many iterations, as described by Terven [66]. Some models, aimed at increased performance for edge devices, take advantage of quantization. The third approach is *knowledge distillation*—training a smaller student model using a larger teacher or multiple models. There are many more approaches, described in greater detail in the following surveys [65,67,68].

In Level 4 edge intelligence, training and inference are performed cooperatively between the edge and the cloud. With some of the training occurring close to the data source, the variety of data can increase. The edge node can also anonymize the data during training before being sent to the cloud. Xu et al. [69] proposed two approaches to cloud–edge collaboration. Both approaches use network splitting, which was mentioned in the Level 1 EI, by dividing the network between the edge and the cloud. The edge contains one randomly selected controller node, which communicates with the cloud, and agent nodes, which communicate with the controller node. The split is selected according to multiple criteria: throughput, energy consumption, processing speed, and available memory. In the first approach, which is more akin to Level 1 EI, the model is trained in the cloud and then deployed to the edge via the master node and to the cloud. The abovementioned algorithm splits the network. The second approach is more complex, as the training occurs on both the edge and the cloud. Before the training starts, the network is split and deployed across nodes with initialization parameters. The output from the last layer present on the edge is sent to the cloud, where the rest of the network is located. The cloud then calculates the loss function and adjusts the parameters, which are then sent back to the edge nodes. The authors also employ an early exit strategy by deploying a simplified cloud model to the controller node. This strategy is applied in case of network problems or the incapability of the network to complete the inference in the required time.

Level 5 is the first level not to utilize the cloud resources. Without access to the processing power of the cloud, the training is performed in cooperation between multiple edge nodes. The nodes share and offload the data between each other or offload it to nodes focused solely on training. Galanopoulos et al. [70] implemented this level of EI in a face recognition application. The authors state that their approach is 50% faster and 30% more accurate than competitive approaches while consuming fewer resources. Their approach considers each node's processing capabilities as the nodes are aware of each other's resources and can better select where to offload their processing. The two-stage approach, in which the first step is the feature extraction and the second step is the classification, achieved better results when dividing the steps between two devices. The accuracy of the solution increased by 10% and the delay decreased by 16%. The solution can also be modified to focus on either accuracy or latency.

In Level 6, training and inference occur on a single device. As previously mentioned, the edge comprises various types of devices—SBCs, MCUs, or Mini-PCs. The performance restrictions on some of these devices led to the creation of AI models focused on smaller sizes and faster processing. One interesting area is the adoption of TinyML—machine

learning aimed at use in microcontrollers [71]. Whereas other devices possess an OS, allowing greater freedom in selecting a programming language or platform to work on, MCUs are more restricted. Developing a TinyML model focuses on taking advantage of the selected device's hardware resources. No communication overhead is introduced by placing such a model right at the data source. We would like to draw attention to MCUNet [72], a highly optimized deep learning framework aimed at low-cost MCUs. The authors trained three models: ImageNet, Visual Wake Words, and Speech Commands. In image recognition, their approach achieved a record 70.7% accuracy on a device with 512 kB memory and 2 MB storage. The solution reduced the memory usage by 3.5× when compared to ResNet-18 and MobileNetV2-0.75, other networks created for use in resource-restricted devices. In Visual Wake Words, a dataset created for speech recognition, the network achieved 2.4× faster inference than the previous state of the art. Their works highlight the potential of using very low-power devices for edge intelligence.

The relevant publications are summarized in Table 2 and assigned to each level of edge intelligence.

Table 2. Levels of edge intelligence and relevant publications.

Level	Title	Reference
1	DeepSplit: Dynamic Splitting of Collaborative Edge-Cloud Convolutional Neural Networks	[61]
	Auto-Split: A General Framework of Collaborative Edge-Cloud AI	[62]
2	Towards Efficient Inference: Adaptively Cooperate in Heterogeneous IoT Edge Cluster	[63]
3	Literature Review of Deep Network Compression	[64]
	AI on the edge: a comprehensive review	[65]
	A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS	[66]
	Model Compression for Deep Neural Networks: A Survey	[67]
	Edge Computing Technology Enablers: A Systematic Lecture Study	[68]
4	A collaborative cloud-edge computing framework in distributed neural network	[69]
5	Cooperative Edge Computing of Data Analytics for the Internet of Things	[70]
6	TinyML Meets IoT: A Comprehensive Survey	[71]
	MCUNet: Tiny Deep Learning on IoT Devices	[72]

Many edge intelligence solutions already utilize containers. This section will provide examples and describe how the containers are helping. We will also provide examples in which containers could be a beneficial extension of an already existing solution.

5.1. Federated Learning

The distribution of processes and data has been used in the fields of data processing, data storage, and AI for many years already. Whenever a single computer lacks the resources to solve the task by itself, the process can be split between multiple computers. Traditionally, the computers were abstracted to represent a single system, with the data being split according to their capabilities. While this approach is sufficient in many cases, it does not address data privacy issues. The data are freely shared between computers, which can lead to privacy concerns and network congestion.

Federated learning (FL) serves as the embodiment of edge intelligence, using the advantages of edge computing while simultaneously solving many problems inherent to such an environment. Unlike in the traditional approach to distributed computing, the data are not shared between devices. Instead, each device has access only to its data. Each device is trained on a separate dataset during the model training, and only the resulting model is shared. This approach mitigates or outright eliminates many of the security and privacy concerns. After each training, the models are collected and adjusted according to the selected criteria. The adjusted models are then sent back to each device to serve as a base for the next round of training. An example architecture is shown in Figure 5.

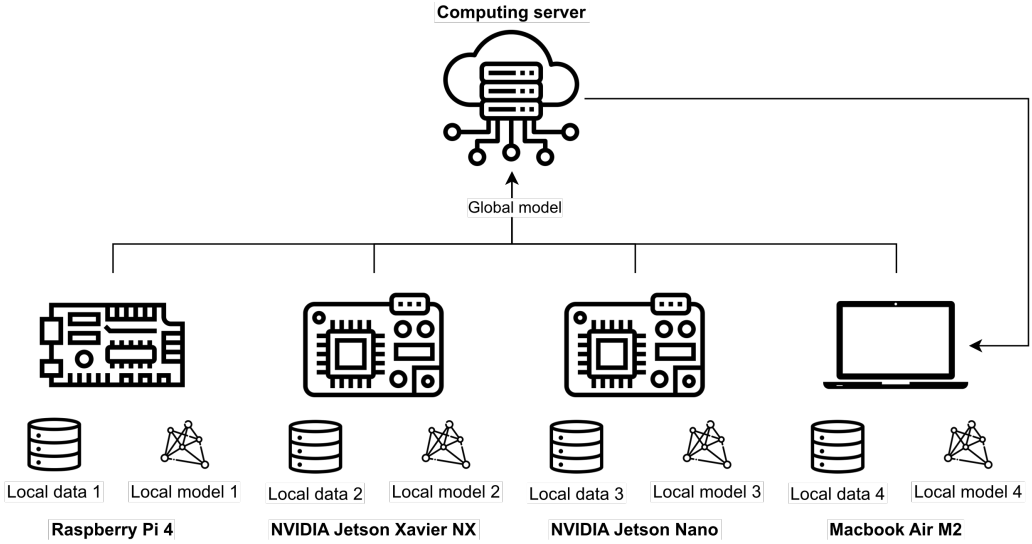

Figure 5. Federated learning architecture.

As the devices collecting the data are the same ones processing it, a new problem arises regarding the availability of the data. In heterogeneous environments, applications may not be compatible with all the available devices. In [73], Brecko et al. describe the most popular FL frameworks along with supported operating systems. They have selected five frameworks, of which only two supported all selected operating systems. Two of the selected frameworks did not support Microsoft Windows, the most popular operating system in consumer desktop computers. The support for different instruction sets, more specifically ARM, was also not found in the two frameworks. This highlights the issue of heterogeneity of devices found in edge environments. As mentioned at the beginning of this section, Docker offers a way to build images compatible with both x86 and ARM architectures, which comprise most edge devices. It is also compatible with the main operating systems—Linux, MacOS, and Windows. As edge computing also deals with mobile devices and smartphones are a great data source, some federated learning frameworks have direct support for their OS. iOS, found in Apple devices, cannot run any containers due to the proprietary nature of Apple's software. On the other hand, Android runs a modified version of the Linux kernel and can therefore be modified. While not officially supported or recommended, it is possible to run Docker containers on an Android device [74]. The steps required include modifying the device kernel but highlighting the potential of containerization in mobile edge computing.

One of the advantages of edge computing is the geolocation of the devices—close to the data source. Due to this, the number of devices available is highly dependent on external factors. There are numerous factors which should be considered when training on a larger number of devices:

- Population density—simply put: more people, more devices. The population density affects the number of available devices in a certain location.
- Country location—people from different countries may behave differently.
- Housing—people living in flats and houses exhibit different behaviors.
- Modes of transportation—stemming from the previously mentioned factors, the modes of transportation.

To address this issue, Chahoud et al. [75] propose the usage of containers. Their proposed solution increases the number of devices usable as the compatibility problem can be circumvented by packaging together all the required software and libraries.

5.2. Robotics

The autonomy of robots is an ever-increasing challenge and an opportunity. The focus has shifted from separating human and robot workers to promoting their collaboration and

cooperation. Artificial intelligence methods are employed to address the challenges arising from this shared environment. The robots now must consider how their actions affect the environment and people around them. Interoperability between different robotic systems can also pose a challenge.

Robot Operating System (ROS) is the standard when it comes to the creation of robotic applications. The abstraction provided by ROS eases the development and integration of software across different platforms. Lumpp et al. [76] propose the utilization of containers in the design flow of robotic applications. The proposed methodology uses Docker, Kubernetes, and ROS to achieve reusable, modular, and portable software components for use in multiple domains. ROS utilizes a graph-based approach to communication where each node represents a process. The authors classify these notes into different hierarchical abstraction levels. The nodes are then clustered into containers to optimize system efficiency. Nodes that require constant communication are, therefore, packaged together to be deployed onto the same device. The authors use ROS-based communication by removing the isolation between the container and the host network. The containers generated also take advantage of image inheritance. Shared packages and libraries are bundled into an image, which is then used as a base for other images. The proposed methodology was then used to design an application simulating an industrial agile production chain. The mobile robot moved pieces from the conveyor belt to the cargo bay. The solution was tested in various configurations: an external server, a main control board, and Nvidia Jetson Nano. The software controlling the robot arms was always deployed on the main control board, and the software for image detection was always deployed on the Nvidia Jetson, as it was the device closest to the camera. The authors then tested and compared the solution on various deployment combinations, with and without the utilization of containers.

5.3. Healthcare

Healthcare and patient care are two of the areas that can benefit the most from edge computing. The data generated in these scenarios is very personal and private. It must, therefore, be kept safe to ensure no malicious actions are possible. The data are also time-sensitive and latency-sensitive, as the patient's condition can change quickly. Network failures and bandwidth congestion can occur at any step on the way to the cloud. These factors can be minimized by moving the data processing from the cloud to the edge. Adopting technologies like IoT, smart devices, and AI has improved the quality and availability of healthcare.

Murphree et al. [77] describe creating a predictive model to calculate the risk of 30-day hospital readmission in patients released from a hospital. The authors are employed at a tertiary care clinic in Rochester, Minnesota, and are familiar with the IT structure present at the facility. The authors highlight the use of many proprietary applications and the use of platform-specific solutions. Another highlighted issue is the difference between research and production environments in a medical setting. For a model to be implemented, it must prove that it can run as well in production as it did during development. Therefore, the model should be as portable as possible without needing modifications between development and production. The authors have combined Docker with REST API applications written in R and Python to address these issues. The model can be deployed anywhere and accessed from other applications by making a simple API call.

Kleftakis et al. [78] describe a fascinating and creative approach to creating an Electronic Health Record using containers. The authors created a digital twin of a patient by dividing parts or organs of the body into containers. The authors applied current approaches to software development by utilizing microservices, containers, orchestrators, and MLOps. The solution is currently aimed for use in cloud-based environments but could be adapted and deployed in edge-based environments, such as hospitals, to enhance data privacy.

5.4. Virtual Assistants

Speech recognition saw giant leaps in the last decade. Personal assistants like Siri, Cortana, and Alexa are already available on mobile devices and vehicles. These assistants utilize a connection to the cloud for speech recognition, leading to privacy concerns. They can perform general tasks, e.g., answering simple questions or controlling smart devices connected to the same network.

Beňo et al. [79] describe an implementation of Microsoft Cognitive Speech Service on the edge utilizing Microsoft Azure services. The solution is made up of two containers. Microsoft provides a container for the Cognitive Speech Service, which the authors deployed on a local device offering sufficient performance. The second container, the voice recognition module, searches for key phrases and executes selected commands. The Cognitive Speech Service container can be deployed on the edge or in the cloud if sufficient performance is not available on the edge. The solution is modifiable and allows for easy set up voice recognition for human-machine interactions. The authors highlight a few advantages of this approach. When deployed on the edge, the response times of the solution were approximately 58% better than when deployed in the cloud. The voice recognition module can be reused in other IoT solutions using services from other providers without much modification. The privacy issue is solved by running the container on the local network where none of the data are sent to the cloud.

5.5. Composite AI

Composite AI combines multiple AI techniques to achieve better results than any single technique by itself. The current approach to AI deals mainly with a single technique, such as large language models, as is the trend today. Initially, ChatGPT expected text input from the users and answered with its own text. However, when combined with other techniques, in this case, speech recognition and speech synthesis [80], the chatbot can perform normal vocal conversations with the user. This approach is underrepresented in edge intelligence as we could not find any existing solutions explicitly aimed at edge computing. Instead, we will provide examples of approaches which could be adapted to edge.

Tara et al. [81] created an ontology model enabling the interoperability of AI agents. Their proposed model consists of data format, agent interaction, and environment. We will focus on the environment as that is where we see containers most beneficial. The authors divide the environment into five parts, which are described in greater detail in the original paper. Clients are the entry point for new data and algorithms and communicate only with the coordinator. The coordinator receives commands from the client, which it then delegates to the components running under it. It also maintains the health of said components. As the authors use blockchain state machines to store events from the lower components, the coordinator is also responsible for this communication. The state is also shared with other coordinators who validate the transaction and store it in their state machine.

The Blockchain State Machine stores information about the location of deployed algorithms, their status, and the location of their assets in the decentralized storage. The Decentralized Storage holds all the files required to start and run the algorithms. Docker images are stored in registries, which are typically centralized, but some extensions and solutions provide a way of decentralizing the registry. The authors also highlight the need for clearly defined input and output definitions, which could be included in the images themselves. The Execution Environment Manager is responsible for resource allocation, environment creation, and running the algorithm inside the created environment. The authors note the usage of VMs or containers when creating these environments, so we would like to expand on it. Typical VMs take a long time to start. If the machine has to be created from scratch, it can take several minutes to perform the required tasks. With containers, it can start in seconds. Docker also natively searches the registry for any images not found locally, which in this case would be the Decentralized Storage, solving the problem of getting the algorithms to the agent.

The Table 3 summarizes the mentioned publications and assigns a level of EI (if applicable) together with a short description. The table contains some additional publications relevant to the reviewed research area.

Table 3. Levels of edge intelligence and relevant publications.

Reference	Year	Application Area	Description	EI Level
[75]	2023	Federated learning	Dynamic FL deployment and learning scheme	4
[76]	2021	Robotics	Design methodology for ROS-based applications	4
[77]	2021	Healthcare	Readmission prediction system for healthcare facilities	3
[78]	2022	Healthcare	Electronic health records decomposing the patient's body into containers	-
[79]	2021	Virtual assistant	Voice control for human-machine interaction	3
[81]	2022	Composite AI	Ontology model for development of multi-agent AI systems	-
[82]	2018	Healthcare	Human activity recognition	3
[83]	2021	Security	Re-identification of people across multiple cameras	2
[84]	2022	Wildfire modelling	A federation architecture to enable a composable infrastructure	-
[85]	2019	Computer vision	Architecture for image processing	3

6. Discussion and Challenges

As stated in Section 2, there are many challenges when it comes to the development and deployment of edge intelligence solutions [12,86–88]. During our research, we have selected a few of them which could be solved or at least helped with the use of containerization.

Edge environments are heterogeneous. The devices use different hardware architectures, most commonly ×86, ARM, and RISC-V, in either a 32-bit or 64-bit version. The differences between these architectures result in an environment where solutions must be modified to accommodate the used architecture. In 2019, Docker officially partnered with ARM to allow for easy-to-use cross-platform image creation, which results in a single image deployable across multiple architectures.

Software compatibility refers to the different operating systems used in edge devices. While Linux is the most used OS in both cloud and edge environments, the differences between its distributions pose a challenge. Some AI libraries or applications are only compatible with a few distributions, and running them on anything else requires a long and difficult setup. The problem is exacerbated by the incompatibility of hardware and an OS-specific distribution. These challenges were analyzed and discussed by Huang et al. [89].

The problem of hardware placement and software distribution refers to the need for remote access to edge environments. Edge devices can be placed in hard-to-reach or unsafe environments where direct physical access is impossible or dangerous. Therefore, the solutions should be able to be deployed remotely through the Internet or a local network.

Edge environments often consist of tens or even hundreds of devices. Managing devices at this scale becomes impossible without the use of dedicated orchestrators and managers. Container orchestrators like Kubernetes already provide a way of managing multiple devices and environments.

Devices can suffer from network failures or bandwidth issues, leading to reduced solution performance. In case of any problems, the container can be quickly deployed to another device. The same approach can be used in inconsistent data load scenarios. Containers can be deployed to more devices during high load for better load balancing. During low load, containers can be removed from devices to save on energy costs.

7. Conclusions

Containers are an integral part of cloud-based solutions as they solve the issues of portability, consistency, flexibility, and scalability. Some of these issues are more prolific in edge environments. The nodes are heterogeneous and distributed and can suffer from power or network failures. The software can encounter compatibility issues with the OS or the hardware platform itself. Adopting cloud-native approaches may help make the development and management of edge computing solutions easier [90]. Edge intelligence

requires special attention as platforms and libraries used to develop artificial intelligence solutions are complex and difficult to set up correctly. Packaging the correct versions of required libraries with the application itself can help prevent these issues. Due to the challenges present in edge intelligence, researchers have tried to implement cloud-native approaches in various domains.

We focused on the potential of virtualization for deployment and use in practical applications and the frameworks for containerization. We have discussed the supplementary technologies and approaches to empower containerization and their benefits to software development and deployment. We provided existing examples of edge intelligence solutions and platforms utilizing containers. We also discussed the challenges and potential future of containerization and edge intelligence.

This review is a part of our research and development of a container-based system for data processing and artificial intelligence at the edge. We plan on creating a modular system that allows users to create a data processing pipeline by providing a no-code or low-code approach to development.

Author Contributions: Conceptualization and methodology, L.U. and E.K.; original draft preparation, L.U.; review and editing, L.U., E.K., P.P. and I.Z.; supervision, P.P.; project administration and funding acquisition, I.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the APVV grant ENISaC—Edge-eNabled Intelligent Sensing and Computing (APVV-20-0247).

Data Availability Statement: The data presented in this study are available in this article.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application programming interface
AI	Artificial Intelligence
CI/CD	Continuous Integration and Continuous Delivery
EC	Edge Computing
EI	Edge Intelligence
FL	Federated Learning
IaaS	Infrastructure as a Service
IoT	Internet of Things
K8s	Kubernetes
ML	Machine Learning
MCU	Microcontroller Unit
NN	Neural Network
OCI	Open Container Initiative
OS	Operating System
PaaS	Platform as a Service
PC	Personal Computer
ROS	Robot Operating System
SBC	Single Board Computer
SaaS	Software as a Service
VM	Virtual Machine

References

1. Arthur, C. Tech giants may be huge, but nothing matches big data. *The Guardian*, 23 August 2013.
2. Hirsch, D.D. The glass house effect: Big Data, the new oil, and the power of analogy. *Maine Law Rev.* **2013**, *66*, 373.
3. Vailshery, L.S. *Public Cloud Services End-User Spending Worldwide from 2017 to 2024*; Statista: New York, NY, USA, 2023.
4. Berisha, B.; Mëziu, E.; Shabani, I. Big data analytics in Cloud computing: An overview. *J. Cloud Comput.* **2022**, *11*, 24. [CrossRef]

5. Alexander, E. *Essential Internet Traffic Statistics in 2024*; ZipDo: Aichach, Germany, 2023.
6. Quy, N.M.; Ngoc, L.A.; Ban, N.T.; Hau, N.V.; Quy, V.K. Edge Computing for Real-Time Internet of Things Applications: Future Internet Revolution. *Wirel. Pers. Commun.* **2023**, *132*, 1423–1452. [CrossRef]
7. AWS. *What Is Containerization?—Containerization Explained*; AWS: Seattle, WA, USA, 2023.
8. Angel, N.A.; Ravindran, D.; Vincent, P.M.D.R.; Srinivasan, K.; Hu, Y.C. Recent Advances in Evolving Computing Paradigms: Cloud, Edge, and Fog Technologies. *Sensors* **2022**, *22*, 196. [CrossRef]
9. Bourechak, A.; Zedadra, O.; Kouahla, M.N.; Guerrieri, A.; Seridi, H.; Fortino, G. At the Confluence of Artificial Intelligence and Edge Computing in IoT-Based Applications: A Review and New Perspectives. *Sensors* **2023**, *23*, 1639. [CrossRef]
10. Shirazi, S.N.; Gouglidis, A.; Farshad, A.; Hutchison, D. The Extended Cloud: Review and Analysis of Mobile Edge Computing and Fog From a Security and Resilience Perspective. *IEEE J. Sel. Areas Commun.* **2017**, *35*, 2586–2595. [CrossRef]
11. Marinescu, D.C. *Cloud Computing: Theory and Practice*; Morgan Kaufmann: Burlington, MA, USA, 2022.
12. Iftikhar, S.; Gill, S.S.; Song, C.; Xu, M.; Aslanpour, M.S.; Toosi, A.N.; Du, J.; Wu, H.; Ghosh, S.; Chowdhury, D.; et al. AI-based fog and edge computing: A systematic review, taxonomy and future directions. *Internet Things* **2023**, *21*, 100674. [CrossRef]
13. Rosendo, D.; Costan, A.; Valduriez, P.; Antoniu, G. Distributed intelligence on the Edge-to-Cloud Continuum: A systematic literature review. *J. Parallel Distrib. Comput.* **2022**, *166*, 71–94. [CrossRef]
14. Kang, P.; Jo, J. Benchmarking Modern Edge Devices for AI Applications. *IEICE Trans. Inf. Syst.* **2021**, *E104.D*, 394–403. [CrossRef]
15. Hussain, H.; Tamizharasan, P.S.; Rahul, C.S. Design possibilities and challenges of DNN models: A review on the perspective of end devices. *Artif. Intell. Rev.* **2022**, *55*, 5109–5167. [CrossRef]
16. M Computers s.r.o. NVIDIA Jetson. Available online: https://mcomputers.cz/en/products-and-services/nvidia/jetson/ (accessed on 8 January 2024).
17. Microway. Comparison of NVIDIA GeForce GPUs and NVIDIA Tesla GPUs. Available online: https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/ (accessed on 8 January 2024).
18. Radchenko, G.I.; Alaasam, A.B.A.; Tchernykh, A.N. Comparative Analysis of Virtualization Methods in Big Data Processing. *Supercomput. Front. Innov.* **2019**, *6*, 48–79. [CrossRef]
19. Willner, A.; Gowtham, V. Towards a Reference Architecture Model for Industrial Edge Computing. *IEEE Commun. Stand. Mag.* **2020**, *4*, 42–48. [CrossRef]
20. Willner, A. The European Edge Computing Consortium (EECC) Presented the Reference Architecture Model Edge Computing (RAMEC). Fraunhofer FOKUS. 24 October 2019. Available online: https://www.fokus.fraunhofer.de/en/ngni/news/ecf-eecc_2019_12 (accessed on 15 February 2024).
21. Desai, A.; Oza, R.; Sharma, P.; Patel, B. Hypervisor: A survey on concepts and taxonomy. *Int. J. Innov. Technol. Explor. Eng.* **2013**, *2*, 222–225.
22. Schlosser, D.; Duelli, M.; Goll, S. Performance Comparison of Hardware Virtualization Platforms. In *NETWORKING 2011*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6640, pp. 393–405. [CrossRef]
23. Takabi, H.; Joshi, J.B.; Ahn, G.J. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Secur. Priv.* **2010**, *8*, 24–31. [CrossRef]
24. Dhule, C.; Shrawankar, U. Impact Analysis of Hypervisors on the Performance of Virtualized Resources. In *Proceedings of the Integrated Intelligence Enable Networks and Computing*; Algorithms for Intelligent Systems; Singh Mer, K.K., Semwal, V.B., Bijalwan, V., Crespo, R.G., Eds.; Springer: Singapore, 2021; pp. 421–427. [CrossRef]
25. De Lauretis, L. From Monolithic Architecture to Microservices Architecture. In Proceedings of the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Berlin, Germany, 27–30 October 2019; pp. 93–96. [CrossRef]
26. Bhardwaj, A.; Krishna, C.R. Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey. *Arab. J. Sci. Eng.* **2021**, *46*, 8585–8601. [CrossRef]
27. Dua, R.; Raja, A.R.; Kakadia, D. Virtualization vs Containerization to Support PaaS. In Proceedings of the 2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, 11–14 March 2014; pp. 610–614. [CrossRef]
28. Jain, S.M. Namespaces. In *Linux Containers and Virtualization: A Kernel Perspective*; Apress: Berkeley, CA, USA, 2020; pp. 31–43. [CrossRef]
29. Martin, A.; Raponi, S.; Combe, T.; Di Pietro, R. Docker ecosystem—Vulnerability Analysis. *Comput. Commun.* **2018**, *122*, 30–43. [CrossRef]
30. Đorđević, B.; Timčenko, V.; Sakić, D.; Davidović, N. File system performance for type-1 hypervisors on the Xen and VMware ESXi. In Proceedings of the 2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH), East Sarajevo, Bosnia and Herzegovina, 16–18 March 2022; pp. 1–6. [CrossRef]
31. Sharma, P.; Chaufournier, L.; Shenoy, P.; Tay, Y.C. Containers and Virtual Machines at Scale: A Comparative Study. In Proceedings of the 17th International Middleware Conference, Trento, Italy, 12–16 December 2016; Middleware '16; ACM: New York, NY, USA, 2016; pp. 1–13. [CrossRef]
32. Aniruddh, M.; Dinkar, A.; Mouli, S.C.; Sahana, B.; Deshpande, A.A. Comparison of Containerization and Virtualization in Cloud Architectures. In Proceedings of the 2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), Bangalore, India, 9–11 July 2021; pp. 1–5. [CrossRef]

33. Abuabdo, A.; Al-Sharif, Z.A. Virtualization vs. Containerization: Towards a Multithreaded Performance Evaluation Approach. In Proceedings of the 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA), Abu Dhabi, United Arab Emirates, 3–7 November 2019; pp. 1–6. [CrossRef]
34. Watada, J.; Roy, A.; Kadikar, R.; Pham, H.; Xu, B. Emerging Trends, Techniques and Open Issues of Containerization: A Review. *IEEE Access* **2019**, *7*, 152443–152472. [CrossRef]
35. Jabbari, R.; bin Ali, N.; Petersen, K.; Tanveer, B. What is DevOps? A Systematic Mapping Study on Definitions and Practices. In Proceedings of the Scientific Workshop Proceedings of XP2016, Edinburgh, UK, 24 May 2016; XP '16 Workshops; ACM: New York, NY, USA, 2016; pp. 1–11. [CrossRef]
36. Luz, W.P.; Pinto, G.; Bonifácio, R. Adopting DevOps in the real world: A theory, a model, and a case study. *J. Syst. Softw.* **2019**, *157*, 110384. [CrossRef]
37. Team, F.B.U. Continuous Integration and Continuous Deployment. In *Cloud-Native Application Architecture: Microservice Development Best Practice*; Springer Nature: Singapore, 2024; pp. 351–382. [CrossRef]
38. Poth, A.; Werner, M.; Lei, X. How to Deliver Faster with CI/CD Integrated Testing Services? In Proceedings of the Systems, Software and Services Process Improvement, Bilbao, Spain, 5–8 September 2018; Communications in Computer and Information Science; Larrucea, X., Santamaria, I., O'Connor, R.V., Messnarz, R., Eds.; Springer: Cham, Switzerland, 2018; pp. 401–409. [CrossRef]
39. JetBrains. *The State of Developer Ecosystem in 2023 Infographic*; JetBrains: Prague, Czech Republic, 2023.
40. Shu, R.; Gu, X.; Enck, W. A Study of Security Vulnerabilities on Docker Hub. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale , AZ, USA, 22–24 March 2017; pp. 269–280. [CrossRef]
41. Combe, T.; Martin, A.; Di Pietro, R. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Comput.* **2016**, *3*, 54–62. [CrossRef]
42. Chamoli, S.; Sarishma. Docker Security: Architecture, Threat Model, and Best Practices. In *Soft Computing: Theories and Applications*; Advances in Intelligent Systems and Computing; Sharma, T.K., Ahn, C.W., Verma, O.P., Panigrahi, B.K., Eds.; Springer: Singapore, 2021; pp. 253–263. [CrossRef]
43. Tank, D.; Aggarwal, A.; Chaubey, N. Virtualization vulnerabilities, security issues, and solutions: A critical study and comparison. *Int. J. Inf. Technol.* **2022**, *14*, 847–862. [CrossRef]
44. The Linux Foundation. *About the Open Container Initiative*; The Linux Foundation: San Francisco, CA, USA, 2015.
45. Dockerfile Reference. 2024. Available online: https://docs.docker.com/reference/dockerfile/ (accessed on 7 March 2024).
46. Merkel, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.* **2014**, *2014*.
47. Ismail, B.I.; Mostajeran Goortani, E.; Ab Karim, M.B.; Ming Tat, W.; Setapa, S.; Luke, J.Y.; Hong Hoe, O. Evaluation of Docker as Edge computing platform. In Proceedings of the 2015 IEEE Conference on Open Systems (ICOS), Melaka, Malaysia, 24–26 August 2015; pp. 130–135. [CrossRef]
48. Docker Documentation. Review of the Docker Daemon Attack Surface. 2023. Available online: https://docs.docker.com/engine/security/#docker-daemon-attack-surface (accessed on 30 December 2023).
49. Docker Documentation. Run the Docker Daemon as a Non-Root User (Rootless Mode). 2023. Available online: https://docs.docker.com/engine/security/rootless/ (accessed on 30 December 2023).
50. Rahmansyah, R.; Suryani, V.; Arif Yulianto, F.; Hidayah Ab Rahman, N. Reducing Docker Daemon Attack Surface Using Rootless Mode. In Proceedings of the 2021 International Conference on Software Engineering & Computer Systems and 4th International Conference on Computational Science and Information Management (ICSECS-ICOCSIM), Pekan, Malaysia, 24–26 August 2021; pp. 499–502. [CrossRef]
51. Casalicchio, E. Container Orchestration: A Survey. In *Systems Modeling: Methodologies and Tools*; EAI/Springer Innovations in Communication and Computing; Springer International Publishing: Cham, Switzerland, 2018; pp. 221–235. [CrossRef]
52. Nguyen, P. Update Docker Who? By Acquiring CoreOS, Red Hat Aims to Be the Kubernetes Company. *The New Stack*, 5 February 2018.
53. Burns, B.; Beda, J.; Hightower, K.; Evenson, L. *Kubernetes: Up and Running*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2022.
54. Riggins, J.; Williams, A. Will Kubernetes Play a Role in Edge Computing? *The New Stack*, 20 December 2019.
55. Rolon-Mérette, D.; Ross, M.; Rolon-Mérette, T.; Church, K. Introduction to Anaconda and Python: Installation and setup. *Quant. Methods Psychol.* **2020**, *16*, S3–S11. [CrossRef]
56. Conda. Getting Started with Conda. Available online: https://docs.conda.io/projects/conda/en/latest/user-guide/getting-started.html (accessed on 8 March 2024).
57. Mell, P.; Grance, T. The NIST Definition of Cloud Computing. 2011. Available online: http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf (accessed on 22 January 2024).
58. OpenFog Consortium Architecture Working Group. OpenFog Reference Architecture for Fog Computing, 2017. Available online: https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf (accessed on 22 January 2024).
59. Zhou, Z.; Chen, X.; Li, E.; Zeng, L.; Luo, K.; Zhang, J. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. *Proc. IEEE* **2019**, *107*, 1738–1762. [CrossRef]
60. Bilan, M. Statistics of ChatGPT & Generative AI in Business: 2024 Report. 2023. Master of Code. Available online: https://masterofcode.com/blog/statistics-of-chatgpt-generative-ai-in-business-2023-report (accessed on 2 January 2024).

61. Mehta, R.; Shorey, R. DeepSplit: Dynamic Splitting of Collaborative Edge-Cloud Convolutional Neural Networks. In Proceedings of the 2020 International Conference on COMmunication Systems & NETworkS (COMSNETS), Bengaluru, India, 7–11 January 2020; pp. 720–725. [CrossRef]
62. Banitalebi-Dehkordi, A.; Vedula, N.; Pei, J.; Xia, F.; Wang, L.; Zhang, Y. Auto-Split: A General Framework of Collaborative Edge-Cloud AI. *arXiv* **2021**, arXiv:2108.13041.
63. Yang, X.; Qi, Q.; Wang, J.; Guo, S.; Liao, J. Towards Efficient Inference: Adaptively Cooperate in Heterogeneous IoT Edge Cluster. In Proceedings of the 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), Washington, DC, USA, 7–10 July 2021; pp. 12–23. [CrossRef]
64. Alqahtani, A.; Xie, X.; Jones, M.W. Literature Review of Deep Network Compression. *Informatics* **2021**, *8*, 77. [CrossRef]
65. Su, W.; Li, L.; Liu, F.; He, M.; Liang, X. AI on the edge: A comprehensive review. *Artif. Intell. Rev.* **2022**, *55*, 6125–6183. [CrossRef]
66. Terven, J.; Córdova-Esparza, D.M.; Romero-González, J.A. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Mach. Learn. Knowl. Extr.* **2023**, *5*, 1680–1716. [CrossRef]
67. Li, Z.; Li, H.; Meng, L. Model Compression for Deep Neural Networks: A Survey. *Computers* **2023**, *12*, 60. [CrossRef]
68. Douch, S.; Abid, M.R.; Zine-Dine, K.; Bouzidi, D.; Benhaddou, D. Edge Computing Technology Enablers: A Systematic Lecture Study. *IEEE Access* **2022**, *10*, 69264–69302. [CrossRef]
69. Xu, S.; Zhang, Z.; Kadoch, M.; Cheriet, M. A collaborative cloud-edge computing framework in distributed neural network. *EURASIP J. Wirel. Commun. Netw.* **2020**, *2020*, 211. [CrossRef]
70. Galanopoulos, A.; Salonidis, T.; Iosifidis, G. Cooperative Edge Computing of Data Analytics for the Internet of Things. *IEEE Trans. Cogn. Commun. Netw.* **2020**, *6*, 1166–1179. [CrossRef]
71. Dutta, D.L.; Bharali, S. TinyML Meets IoT: A Comprehensive Survey. *Internet Things* **2021**, *16*, 100461. [CrossRef]
72. Lin, J.; Chen, W.; Lin, Y.; Cohn, J.; Gan, C.; Han, S. MCUNet: Tiny Deep Learning on IoT Devices. *arXiv* **2020**, arXiv:2007.10319.
73. Brecko, A.; Kajati, E.; Koziorek, J.; Zolotova, I. Federated Learning for Edge Computing: A Survey. *Appl. Sci.* **2022**, *12*, 9124. [CrossRef]
74. Oliveira, F. Docker on Android, 2021. Available online: https://gist.github.com/FreddieOliveira/efe850df7ff3951cb62d74bd770dce27 (accessed on 31 December 2023).
75. Chahoud, M.; Sami, H.; Mourad, A.; Otoum, S.; Otrok, H.; Bentahar, J.; Guizani, M. On-Demand-FL: A Dynamic and Efficient Multicriteria Federated Learning Client Deployment Scheme. *IEEE Internet Things J.* **2023**, *10*, 15822–15834. [CrossRef]
76. Lumpp, F.; Panato, M.; Fummi, F.; Bombieri, N. A Container-based Design Methodology for Robotic Applications on Kubernetes Edge-Cloud architectures. In Proceedings of the 2021 Forum on Specification & Design Languages (FDL), Antibes, France, 8–10 September 2021; pp. 1–8. [CrossRef]
77. Murphree, D.H.; Quest, D.J.; Allen, R.M.; Ngufor, C.; Storlie, C.B. Deploying Predictive Models In A Healthcare Environment—An Open Source Approach. In Proceedings of the 2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Honolulu, HI, USA, 18–21 July 2018; pp. 6112–6116. [CrossRef]
78. Kleftakis, S.; Mavrogiorgou, A.; Mavrogiorgos, K.; Kiourtis, A.; Kyriazis, D. Digital Twin in Healthcare Through the Eyes of the Vitruvian Man. In *Innovation in Medicine and Healthcare*; Smart Innovation, Systems and Technologies; Chen, Y.W., Tanaka, S., Howlett, R.J., Jain, L.C., Eds.; Springer: Singapore, 2022; pp. 75–85. [CrossRef]
79. Beňo, L.; Pribiš, R.; Drahoš, P. Edge Container for Speech Recognition. *Electronics* **2021**, *10*, 2420. [CrossRef]
80. OpenAI Blog. Available online: https://openai.com/blog (accessed on 15 February 2024).
81. Tara, A.; Taban, N.; Turesson, H. Performance Analysis of an Ontology Model Enabling Interoperability of Artificial Intelligence Agents. In *Proceedings of the Artificial Intelligence Trends in Systems*; Lecture Notes in Networks and Systems; Silhavy, R., Ed.; Springer: Cham, Switzerland, 2022; pp. 395–406. [CrossRef]
82. Al-Rakhami, M.; Alsahli, M.; Hassan, M.M.; Alamri, A.; Guerrieri, A.; Fortino, G. Cost Efficient Edge Intelligence Framework Using Docker Containers. In Proceedings of the 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech), Athens, Greece, 12–15 August 2018; pp. 800–807. [CrossRef]
83. Chen, C.H.; Liu, C.T. Person Re-Identification Microservice over Artificial Intelligence Internet of Things Edge Computing Gateway. *Electronics* **2021**, *10*, 2264. [CrossRef]
84. Altintas, I.; Perez, I.; Mishin, D.; Trouillaud, A.; Irving, C.; Graham, J.; Tatineni, M.; DeFanti, T.; Strande, S.; Smarr, L.; et al. Towards a Dynamic Composability Approach for using Heterogeneous Systems in Remote Sensing. In Proceedings of the 2022 IEEE 18th International Conference on e-Science (e-Science), Salt Lake City, UT, USA, 11–14 October 2022; pp. 336–345. [CrossRef]
85. Huang, Y.; Cai, K.; Zong, R.; Mao, Y. Design and implementation of an edge computing platform architecture using Docker and Kubernetes for machine learning. In Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications, Xi'an, China, 8–10 March 2019; pp. 29–32. [CrossRef]
86. Zhang, J.; Tao, D. Empowering Things with Intelligence: A Survey of the Progress, Challenges, and Opportunities in Artificial Intelligence of Things. *IEEE Internet Things J.* **2021**, *8*, 7789–7817. [CrossRef]
87. Filho, C.P.; Marques, E.; Chang, V.; Dos Santos, L.; Bernardini, F.; Pires, P.F.; Ochi, L.; Delicato, F.C. A Systematic Literature Review on Distributed Machine Learning in Edge Computing. *Sensors* **2022**, *22*, 2665. [CrossRef]
88. Hoffpauir, K.; Simmons, J.; Schmidt, N.; Pittala, R.; Briggs, I.; Makani, S.; Jararweh, Y. A Survey on Edge Intelligence and Lightweight Machine Learning Support for Future Applications and Services. *J. Data Inf. Qual.* **2023**, *15*, 20. [CrossRef]

89. Huang, K.; Chen, B.; Wu, S.; Cao, J.; Ma, L.; Peng, X. Demystifying Dependency Bugs in Deep Learning Stack. *arXiv* **2023**, arXiv:2207.10347.
90. Surianarayanan, C.; Chelliah, P.R., Delineating Cloud-Native Edge Computing. In *Essentials of Cloud Computing: A Holistic, Cloud-Native Perspective*; Springer International Publishing: Cham, Switzerland, 2023; pp. 347–357. [CrossRef]

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