future internet

MDPI

Article

Implementing Internet of Things Service Platforms with Network Function Virtualization Serverless Technologies

Mauro Femminella [1,2,†] and Gianluca Reali [1,2,*,†]

[1] Department of Engineering, University of Perugia, via G. Duranti 93, 06125 Perugia, Italy; mauro.femminella@unipg.it
[2] Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), 43124 Parma, Italy
* Correspondence: gianluca.reali@unipg.it
† The authors contributed equally to this work.

Abstract: The need for adaptivity and scalability in telecommunication systems has led to the introduction of a software-based approach to networking, in which network functions are virtualized and implemented in software modules, based on network function virtualization (NFV) technologies. The growing demand for low latency, efficiency, flexibility and security has placed some limitations on the adoption of these technologies, due to some problems of traditional virtualization solutions. However, the introduction of lightweight virtualization approaches is paving the way for new and better infrastructures for implementing network functions. This article discusses these new virtualization solutions and shows a proposal, based on *serverless computing*, that uses them to implement container-based virtualized network functions for the delivery of advanced Internet of Things (IoT) services. It includes open source software components to implement both the virtualization layer, implemented through Firecracker, and the runtime environment, based on Kata containers. A set of experiments shows that the proposed approach is fast, in order to boost new network functions, and more efficient than some baseline solutions, with minimal resource footprint. Therefore, it is an excellent candidate to implement NFV functions in the edge deployment of serverless services for the IoT.

Keywords: Internet of Things; micro virtual machines; network function virtualization; serverless computing; performance evaluation

Citation: Femminella, M.; Reali, G. Implementing Internet of Things Service Platforms with Network Function Virtualization Serverless Technologies. *Future Internet* **2024**, *16*, 91. https://doi.org/10.3390/fi16030091

Academic Editors: Panagiotis Papageorgas, Dimitrios Piromalis and Dionisis Kandris

Received: 6 February 2024
Revised: 4 March 2024
Accepted: 5 March 2024
Published: 8 March 2024

1. Introduction

The Internet of Things (IoT) is a network model that refers to the connection of everyday objects to the Internet. These devices can be smartphones, wearables, cameras, home appliances and drones. IoT devices can serve different types of applications, such as entertainment, home automation and industrial functions. Furthermore, a distinction must also be made regarding the mobility of devices. For example, they could serve vehicular applications and the delivery of goods via drones. The vastness of the types of IoT services that can be implemented poses different challenges to the underlying network and computing infrastructure. First of all, it is necessary to make a distinction between the expected performance on the user plane and on the control plane. For example, this is reflected in different specifications in terms of latency in data exchange.

For many applications, the volume of data generated requires the use of technologies that allow their collection, management and processing in an efficient and scalable way. The strategic importance of IoT has therefore stimulated research and standardization activities. For example, the 3rd Generation Partnership Project (3GPP) Release 13 specification introduced *enhanced machine type communication* (eMTC) as a low-power technology that enables IoT through extended coverage, achieved through low-complexity devices and the use of existing LTE (long term evolution) base stations. Although the IoT devices

can sporadically transmit and receive a relatively small volume of data, the supported density of a million of devices per square kilometer, which is tenfold higher than the current maximum density in 4G LTE networks, is a challenging objective for 5G networks [1]. For this reason, the 3GPP Release 17 introduced RedCap (Reduced Capability [2]), aimed at exploiting the proliferation of *massive machine type communication* (mMTC). It ensures the continuation of the service of legacy technologies, such as NB-IoT, by integrating them with new and improved services for different types of devices. In this way, RedCap enables the deployment of IoT services that require increased data exchange speed with respect to mMTC, ensuring coexistence with higher data rate technologies such as the *enhanced mobile broadband* (eMBB), simultaneously reducing user experience complexity and device size, thus improving power-saving aspects. Looking to the future, 6G is expected to give a significant boost to the development of the IoT [3]. Finally, the growing diffusion of AI technologies which learn from data collected by IoT devices could lead to further needs and challenges for system architectures.

The success of virtualization technologies, based either on virtual machines (VMs) or containers, has stimulated the telecommunications industry to pursue a new architectural design of network node functions, named *network function virtualization* (NFV) [4]. It exploits virtualization to deploy nodes (virtual network functions, VNFs) as elementary blocks that can be interconnected to implement complex communication services such as routers, firewalls, software-defined network (SDN) controllers and so on. An NFV platform must satisfy the following general requirements in order to effectively support virtualized functions deployment [5]: it must be easily scalable, with minimal configuration and management effort, and make efficient use of resources while maintaining a secure environment where services are isolated from each other. Many of the current IoT challenges are related to scalability and quality of service (QoS) provisioning. In this context, NFV provides the elasticity needed to deploy additional network functions, for example when QoS tends to deteriorate or when users need additional services.

Regardless of the specific application, modern IoT services are significantly complex and may require the integration of components from different service categories. For example, in connected vehicle scenarios, mMTC services could be integrated with eMBB components, which are characterized by large bandwidth for supporting broadband services, such as video streaming required by remote driving applications. They could also include *ultra-reliable, low-latency communication* (uRLLC) components for introducing safety components, featuring very low communication latency. In order to fulfil their performance requirements, these services may need to be executed, not only in a central, remote cloud, but more and more often in what is known as the edge-to-cloud continuum [6], also leveraging computational capabilities that are closer to the end-users/IoT devices; that is, in edge nodes. This paradigm enables the dynamic processing of data in different locations on the basis of service requirements, current workload and available service capacity. Classic virtualization techniques, such as virtual machines (VMs) and containers, do not always suit the efficiency and promptness needed for the edge-to-cloud continuum, in terms of resource footprint and startup time to launch a service in any potential node of the network to support dynamic and intermittent applications. Differently, serverless computing is a very promising implementation solution for IoT services. In fact, serverless computing is a model for cloud computing in which the cloud provider dynamically allocates only the computation and storage resources needed to execute a particular piece of code (i.e., function), and charges the users consequently using a (*pay-as-you-go* model). In this model, applications are built and deployed as a group of stateless functions, and instantiated on demand only for the needed time (function-as-a-service model, FaaS). The name "serverless" simply indicates that it allows developers to focus only on the application logic, leaving the burden of managing the underlying back-end servers to the cloud provider and making them invisible to the developers. Therefore, if we consider IoT services, serverless computing turns out to be an excellent solution for many of them, for two main reasons. First, IoT services are often characterized by high variable traffic patterns, including bursty, repetitive

and/or even single transmission behavior. IoT applications are intrinsically based on event management and require scalability. Since serverless technologies are event-oriented, this makes them an ideal choice for IoT applications. The promptness and agility of serverless computing is ideal to serve these variable and not predictable patterns, easily supporting sudden spikes in device growth and ultra-high volumes of data, thus offering a natural solution for speed, scalability and availability needs of mMTC IoT services. Second, many IoT services, such as vehicular ones, even without requiring uRLLC, may require/benefit from fast response times, which can be effectively provided from edge nodes. Furthermore, serverless computing can be efficiently used to deploy these services in edge nodes due to its features of promptness, scalability and a low resource footprint. In fact, only the resources needed for current computing tasks are allocated for the time strictly necessary.

Considering this complex scenario, the objective of this paper is to present a service architecture for IoT services that leverages serverless computing and novel NFV technologies. This architecture is suitable for edge deployment, with the aim of mitigating some issues typical of the IoT world, namely resource consumption, service responsiveness and performance (latency, request loss, etc.). Since our proposal involves serverless technologies combined with NFV, it also allows VNF designers to focus on the application's logic only, without having to deal with server management and autoscaling functions, which are provided by the underlying serverless platform, being a distinctive characteristic of the FaaS model. Since the serverless model can ensure that the amount of resources consumed by an IoT application is dynamically controlled and is tailored to the ongoing computing needs, this clearly provides service providers with a valuable feature for implementing IoT for short and on-demand tasks, in particular when devices are silent for most of the time, and transmit short data packets periodically or randomly. However, this scalable approach with enhanced responsiveness to load variation may cause some latency issues, which are analyzed in this paper.

In summary, the main contributions of this paper are as follows:

- We present an analysis of some key IoT use cases, discussing the suitability of implementing NFV through serverless computing functions, analyzing service requirements and matching them with serverless capabilities.
- We present a specific architecture for serverless computing. It is based on the integration of Firecracker and Kata containers, and it is capable of running both classic containerized applications and new packet processing applications, with minimal overhead. It is realized by means of NetBricks, a novel NFV framework optimized for speed of execution and rapid code development.
- We show the results of an experimental campaign to evaluate the overall costs and performance of the proposed serverless framework, which is based exclusively on open source software, discussing its suitability with respect to IoT service requirements.

The paper is organized as follows. In Section 2, we provide some background on serverless computing and previous research that used surveys, dealing with adoption of serverless computing for IoT services. In Section 3, we analyze some IoT scenarios, highlighting different service requirements that are useful to evaluate the possibility of using the serverless approach for their implementation. Section 4 presents a NFV architecture based on these technologies. Its expected performance is analyzed in Section 5, making use of both results from the literature and lab experiments. Finally, Section 6 reports our conclusions.

2. Background and Related Work

2.1. Serverless Computing

Serverless computing is a cloud service model which leverages the elastic capabilities of cloud networks. It allows developers to implement and deploy applications as a dynamic composition of stateless service components. These service components can take the form of microservices or functions. The latter require applications to be developed through functional programming. However, with the related implementation effort, the selective

activation, as needed, of only the functions that are necessary for the execution of the service is achieved, thus minimizing the computational footprint and related costs. In particular, the computing resources needed to execute customers' services, along with the relevant configuration and management, are dynamically provisioned by the serverless platform at runtime through an event-driven approach, named *function as a service* (FaaS). FaaS is different from traditional cloud provisioning methods, in which the amount of computing resources to be used is planned during application design and deployment phases. It takes the concept of pay-as-you-go to the extreme, with a scalability in the management of resources and service requests much higher than that achievable in the management of VMs or classic microservices.

In serverless computing, the cloud service provider is responsible for provisioning, maintaining and configuring the computing infrastructure, consisting of servers. This relieves developers from a significant management burden, allowing them to focus on the application logic only, organized on individual functions executed in containers or microVMs.

The typical architecture of a serverless computing solution is depicted in Figure 1. It is composed of the following three main functional elements:

- An authentication service: The serverless customers use the API provided by the cloud service provider to authenticate and be authorized to use its services. Typically, this is handled via a security token.
- FaaS engine: This is the core of the serverless solution. It gives developers the ability to build, run, deploy and maintain applications without thinking about server infrastructure. Each application is divided into a number of functions. Each function performs a simple task, which runs only when necessary for the service. This also allows for the sharing of different functions among different services. The access to the function pool is managed through an application engine named *API Gateway*, which triggers the execution of the correct function based on events.
- Data storage: although applications are developed and managed according to the FaaS paradigm, users' data need to be stored in a database, which is an essential component of the overall solution.

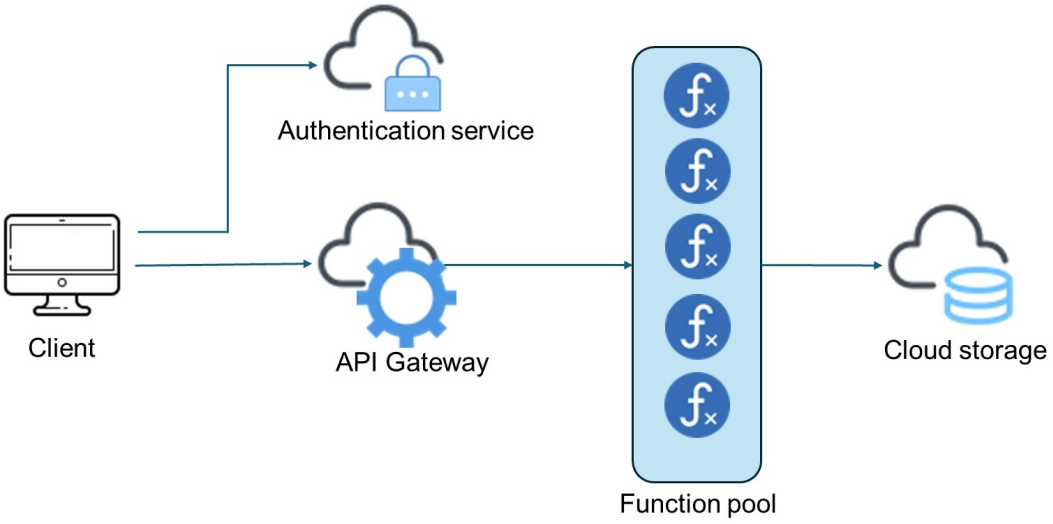

Figure 1. General scheme for serverless computing.

The typical lifecycle of a function, including how it is written and executed in the serverless computing platform, is as follows:

1. The developer writes a function and pushes it to the platform. The developer defines also the events that trigger the execution of the function. The most common event is an HTTP request.

2. The event is triggered. In case of a HTTP request, a remote user triggers it via a click, or another IoT function/device sends a restful message to the target function.
3. The function is executed. The cloud provider checks whether there is an instance of the function already operational, ready to be reused. If not, it spawns a new instance of the function.
4. The result is sent to the requesting entity/user, which will likely be used within the composite application workflow.

Although initially designed to instantiate classic cloud services (we consider the AWS Lambda, Microsoft Azure Functions, Google Cloud Functions and IBM Cloud Functions platforms to be the main serverless cloud providers), serverless computing has also found its place in edge computing, which is used to reduce latency and avoid the consumption of unnecessary energy and resources. In IoT systems, the serverless approach can be particularly useful in the development of back-end components of services, running on edge or cloud nodes.

In serverless computing, container startup latency to instantiate new functions is a well-known problem, which falls under the name *cold start* [7]. A cold start occurs when, in order to satisfy a new user request, it is necessary to start a new container, configure the runtime environment and deploy the requested function. As reported in Ref. [7], typical values of cold startup times ranges from hundreds of ms up to a few seconds. Therefore, cold starts can significantly affect the responsiveness of applications and, consequently, the user experience, making the serverless approach not compliant with the requirements of some types of services. A typical solution for this issue includes keeping a number of instances in the idle state (*warm start*), since reusing an existing function instance would take much less time than handling a new request from scratch. However, this clearly has a non-negligible cost, especially when the types of possible functions increase, as is increasingly common, since the modern way of implementing services consists of breaking them down into a series of micro-functions, each performing a simple task. In fact, in this way, the maintenance, upgradability, scalability and reusability of the software are greatly improved. Alternatively, there are a significant number of proposals exploring the use of artificial intelligence (AI) techniques [8], such as reinforcement learning [9], trying to promptly anticipate the occurrence of a new request and thus limiting the impact of keeping the instance in an inactive state, ready to serve new requests. However, these approaches rely on statistical properties; therefore, they may overestimate the number of idle instances that need to be generated to satisfy hypothetical future requests (over-provisioning and, therefore, constituting a waste of resources) or underestimate them, leading to cold start events.

2.2. Serverless Technologies for the Internet of Things

There is growing interest in using serverless technologies for implementing IoT applications. Cassel et al., in Ref. [10], surveyed 60 papers dealing with serverless computing for IoT, analyzing a number of interesting aspects. Their findings were supported by a rigorous bibliometric analysis. An interesting result was that, in most cases, serverless technologies were used to implement back-end services in edge nodes or in the core cloud section, and only in a limited number of cases were uses found in the part commonly referred to as fog, i.e., the other nodes between the IoT devices and the cloud core. This is perfectly in line with our proposal, targeting edge nodes. Furthermore, they analyzed the most common software technology used to implement service functions, which mainly consisted of containers [7,11–13] and, in a few other cases, Calvin [14], Wasm [15], Unikernels [16], JVM, and IotPy. However, they did not consider new virtualization technologies, such as Kata containers and Firecracker, which are the bricks on which our solution is built. Finally, they provided a general taxonomy that included a large number of aspects, such as potential applications, protocols used to exchange data between IoT applications, and various challenges.

Another paper [14] analyzed the usage of the Calvin platform in edge devices. By using a so-called *actor model*, and with interactions with IoT devices based exclusively on REST APIs, the authors illustrated Kappa, a solution able to include IoT devices in the overall service provisioning. However, the limitation of using HTTP was significant, as also recognized by the authors themselves. In addition, the development of the Calvin application environment is no longer maintained.

The solution presented in Ref. [11] made use of the Knative platform to deploy serverless computing. The analysis showed the impact of cold starts on IoT services, and demonstrated that this can be mitigated via traffic prediction with prefetching. However, Knative appeared to be unsuitable for the IoT environment, due to its excessive overhead.

Other papers focused more on where to place functions when using serverless for IoT. The work in [17] presented an analysis of the criteria used to place functions in a fog computing network of local IoT devices. Edge-to-cloud continuums were explored in other papers. For instance, the paper in [15] presented a solution based on WebAssembly (Wasm), coupled with the usage of the MQTT protocol, which has a lower overhead than HTTP. It leveraged the Zenoh software technology, which makes the overall approach "data-centric". In fact, it allowed for the implementation of a base layer, by creating a geo-distributed storage in the edge-to-cloud continuum. In particular, a key–value pattern was used for the retrieval of information spread throughout the network, modeled as a directed graph. A dataflow programming model was useful in describing the composition of applications over these graphs. Finally, the use of Wams technology was motivated by its small resource footprint for deploying stack-based virtual machines, with the goal of running portable code using an efficient compiler and a lightweight runtime. SERVERLEss4IoT [18] was another solution targeting the edge-to-cloud continuum. The authors of Ref. [18] presented a platform, including IoT devices, for designing, deploying and maintaining applications over the edge-to-cloud continuum. They proposed using a domain-specific modelling language to realize a unified abstraction for specifying the deployment of IoT applications, including serverless functions. However, the paper is more focused on software design and the deployment of software components on IoT devices than on lightweight virtualization solutions for serverless computing in edge nodes.

In summary, we present a comparison between the technologies analyzed in this section. We consider the following features to be of fundamental importance for IoT services:

- Resource footprint;
- Startup time;
- Supported network protocols for function interactions;
- Application portability;
- Security.

From the analysis of these features, which is presented in Table 1, it is clear that, for IoT applications, VMs are not suitable at all. Otherwise, traditional container technologies, such as those using Containerd runtime [19], can be used for applications that do not require frequent startups, thus preventing their usage on most mMTC services with high intermittent profiles. Solutions based on Calvin should be limited to only using HTTP; even if this is the most used protocol, it is a significant limitation. As for Wasm, its internal structure favors low startup times over execution efficiency, which could increase service latency. Finally, Unikernels focuses on resource efficiency and startup speed, but requires the complete rewriting of the app code and even kernel customization, which represents a significant burden for developers willing to go serverless, but relieves them from the management of the platform. Therefore, the conclusion is that a solution that can satisfy all the requirements is still missing, and our proposal is a good candidate for that role.

Table 1. Comparison of the most advanced frameworks for implementing IoT services.

Solution	Resource Footprint	Startup Time	Supported Protocols	Application Portability	Security
VMs	Large: full operating system virtualization	Order of minutes	Any	Complete	Isolation granted, but potentially large attack surfaces
Traditional containers *	Medium: multiple abstraction layers and context switches	Order of seconds [20]	Any	Complete	Isolation issues, caused by the use of a shared kernel (see also Ref. [21])
Calvin [14]	Limited memory overhead and optimized network functions	Potentially low	HTTP only	Only apps using HTTP and specifically built for Calvin	N/A
Wasm [15]	Low footprint	Fast scaling, but execution can be slowed down [10]	Any	Portable code written in high-level languages	Sandboxed environment
Unikernel [16]	Very low, designed for resource-constrained devices	Fast startup time ($\ll$100 ms)	Any (but required libraries may need to be rewritten)	Complete app rewrite is needed	Potentially dangerous kernel functions can be invoked from applications
Proposed	Limited memory overhead and CPU consumption (shown in Section 5.2)	$\leq$100 ms (shown in Section 5.2)	Any	Complete: optimized VNFs written with NetBricks or standard containerized apps	Minimal attack surface: Kata container and microVM isolation from Firecracker

* Container runtime technologies like Containerd, managed by KNative [11], OpenFaaS [12] or OpenWhisk [13] serverless platforms.

3. Key Internet of Things Scenarios

This section illustrates the main features of key IoT scenarios, with the aim of using them in subsequent sections to analyze the suitability of serverless computing platforms for their implementation. The selected scenarios highlight different service levels and performance requirements, generating new business models for mobile operators and their customers [22]. Most of them appear also in the above-mentioned review paper [10], although with slightly different names. The main advantages of serverless computing over other cloud computing paradigms when used to deploy IoT services lie in speed, high resource efficiency and excellent scalability. We map these features onto the following selected IoT service categories.

3.1. Internet of Vehicles

Internet of Vehicles (IoV), also referred to as Vehicle to Everything (V2X), is a scenario that includes different interaction modes. They include Vehicle to Pedestrians (V2P), allowing user devices to communicate with in-vehicle devices, Vehicle to Infrastructure (V2I), by which in-vehicle devices can communicate with roadside infrastructure, and Vehicle to Network (V2N), enabling in-vehicle devices to exchange data with edge–cloud platforms, in addition to the classic Vehicle to Vehicle (V2V) communication. This scenario is part of the 3GPP Cellular V2X (C-V2X) service, which includes both LTE-V2X (3GPP TR 22.885) and 5G V2X for automated driving (3GPP TR 22.886). The latter poses significant performance challenges for realizing future transport services, such as vehicle platooning and remote driving. Furthermore, the expected future urban traffic management systems will rely on the acquisition of a comprehensive traffic picture using massive IoT devices, which will enable multi-source data analysis for decision making based on AI. Some of these

services can be classified as mMTC, whereas others, which have real-time requirements, are better classified as uRLLC. Performance guarantees depend on the supported vehicular functions and include a maximum end-to-end (E2E) latency for sensor information sharing of 3 ms to 1 s, 90 to 99.999% reliability, and a throughput of 10–100 Mbps over a coverage radius lower than 1000 m. These functions are related to different services, such as video streaming or C-V2X messaging [23]. Entertainment services in vehicles and traffic management applications are clearly more tolerant to latency, even if video streaming can be a component of a more complex service, such as remote driving. In general, by analysing service requirements by using the data reported in Ref. [23], the results show that most of services based on V2N communications, which are those of interest for serverless computing deployed in edge nodes, require a maximum data rate equal to 25 Mbps.

Serverless computing could be a great option for implementing IoV mMTC services, especially those dedicated to information sharing, such as those involved in remote vehicle monitoring, cooperative awareness and platooning. Since all of these services have latency requirements in the order of hundreds of ms, using serverless technologies to deploy IoV services at the edge saves computing resources for other, more demanding always-on services. The timeliness in instantiating the necessary functions, typical of serverless computing, allows these functions to be spawned on demand, for example when a new message has to be ingested or processed. Moreover, its scalability allows for responses to sudden surges in demand. On-demand video streaming for vehicular services could also benefit from the scalability and responsiveness of serverless computing.

3.2. Smart Manufacturing

Smart manufacturing is an important recipient of mMTC. Application needs are evolving from those supported by the so-called Narrowband Internet of Things (NB-IoT) and LTE machine-type communication (LTE-M), which were essentially used for sensor data transfer. Present trends include massive sensor deployment, cameras, tablets and wearables. The relevant communication needs include data acquisition from much more differentiated sources, based on human–machine interaction, and even augmented reality. The technology evolution is converging towards throughput in the order of 10 Mbps and, in some specific services with real time requirements, a maximum latency of 2–3 ms, compliant with uRLLC requirements.

Furthermore, in this case, serverless computing, with its responsiveness and limited resource footprint, can be a suitable solution for implementing mMTC in an industrial environment. The ability to deploy services on edge nodes helps in limiting the overall latency in message delivery and processing. However, serverless computing is not a one-size-fits-all solution and may not be suitable for services that require very low latency, such as remote robot control.

3.3. Smart Health

New healthcare services leveraging the IoT in hospitals can include patient positioning, continuous monitoring of vital signs, such as respiration and heartbeat, nursing applications, remote control of infusion pumps and robotic assistance. In some cases, these mMTC services can be deployed in conjunction with uRLLC services, such as remote ultrasonography and remote surgery. Some IoT services may require the exchange of operation and control data transfer of high-resolution medical images and videos from cameras and medical instruments, with data rates between 1 and 20 Mbps in both up and down links (eMBB requirements) and latency values in the range 20–100 ms. In the cases of services not strictly related to emergency management, such as the remote monitoring of patients, latency requirements can be relaxed. In these cases, thanks to its excellent responsiveness, serverless computing can represent an efficient way to deploy healthcare services across the entire edge-to-cloud continuum. In fact, the instantiation of specific functions, such as alert mechanisms for medical personnel, can be activated promptly by specific events, such as the receipt of health data records.

3.4. Smart Grid

Energy production and transport will highly benefit from advanced IoT for the efficient exploitation of intermittent renewable energy sources. Since their random nature makes power balance and control difficult, the distribution network needs to leverage active control energy flow management to address emerging customer needs. Furthermore, energy transportation requires highly efficient processes to achieve high equipment utilization and low line losses, through the accurate regulation of the grid load.

In this environment, it is possible to identify two service types with different performance requirements. The first type consists of online monitoring through sensors. Typically, it requires E2E latency values in the order of seconds and intermittent single-link bandwidth values in the order of tens of kbps. The second type consists of comprehensive video surveillance of transmission lines through fixed cameras and/or unmanned aerial vehicle patrolling. In this case, an uplink bandwidth of up to 10 Mbps and latency values of $\leq$200 ms could be necessary. For both service types, availability values should be at least 99.9%, with a deployment density of 10 information sources (sensors and cameras) for each km^2. It is clear that the first type of service is better suited to serverless computing, since it would be able to exploit all its features: promptness, efficiency and scalability. However, video surveillance requirements do not preclude the adoption of serverless capabilities for these services too, especially if performed non-continuously and triggered by specific events, e.g., associated with the output of the sensor monitoring functions.

4. New Virtualization Solutions for Network Functions

The deployment of mMTC solutions in softwarized networks, like 5G and 6G networks, is expected to benefit from the dynamic management of VNFs and network slicing capabilities [24,25]. Network softwarization indicates new network architectures and technologies that decouple the software implementing network functions, protocols and services from the hardware used to run them, and can be considered the evolution of technologies like NFV and SDN. VNFs used in different IoT applications are deployed in the scope of the corresponding slice. If edge nodes are involved in service provisioning, some of the VNFs used in that situation are expected to be executed in the relevant multi-access edge computing (MEC) section of the slice. Typical examples include VNFs for event management and data exchange in sensor networks [26] and for implementing IoT gateways. In particular, the latter implements strategic functions characterizing individual scenarios, such as data pre-processing and adaptation, load balancing, flow aggregation and, sometimes, slice termination. Its operation must be characterized by scalability and very high availability.

We illustrate new VNF deployment solutions supporting the IoT scenarios presented in Section 3. The general architecture of the IoT system is shown in Figure 2, where the proposed NFV infrastructure consists of a lightweight, security-focused virtualization solution. It aims to smoothly deploy the softwarized network functions, needed by different applications related to the aforementioned scenarios, in a serverless way. The reason for introducing new solutions is that the traditional approach based on VMs is notoriously slow and over-provisioned. Cloud-native network function (CNF) provisioning, based on containers to implement VNF, offers improved performance. However, it is affected by some security and isolation issues, caused by the use of a shared kernel, along with a long cold startup latency [5]. Unikernels, which are minimalist runtime environments running on top of virtual hardware abstractions, are proposed as an alternative to containers, since they are more secure and often faster. However, removing the concept of process from its monolithic appliances, Unikernels gain less flexibility and applicability since dynamic forking, which is a basis for the common multi-process abstraction of conventional UNIX applications, is not supported. Moreover, runtime management, such as online library updating and address space randomization, is not allowed, which complicates debugging. The Unikernels peculiarity of running their applications directly in the kernel ring, in the

same address space, could also entail some security issues, since privilege separation is impractical and dangerous kernel functions could be invoked from applications [27,28].

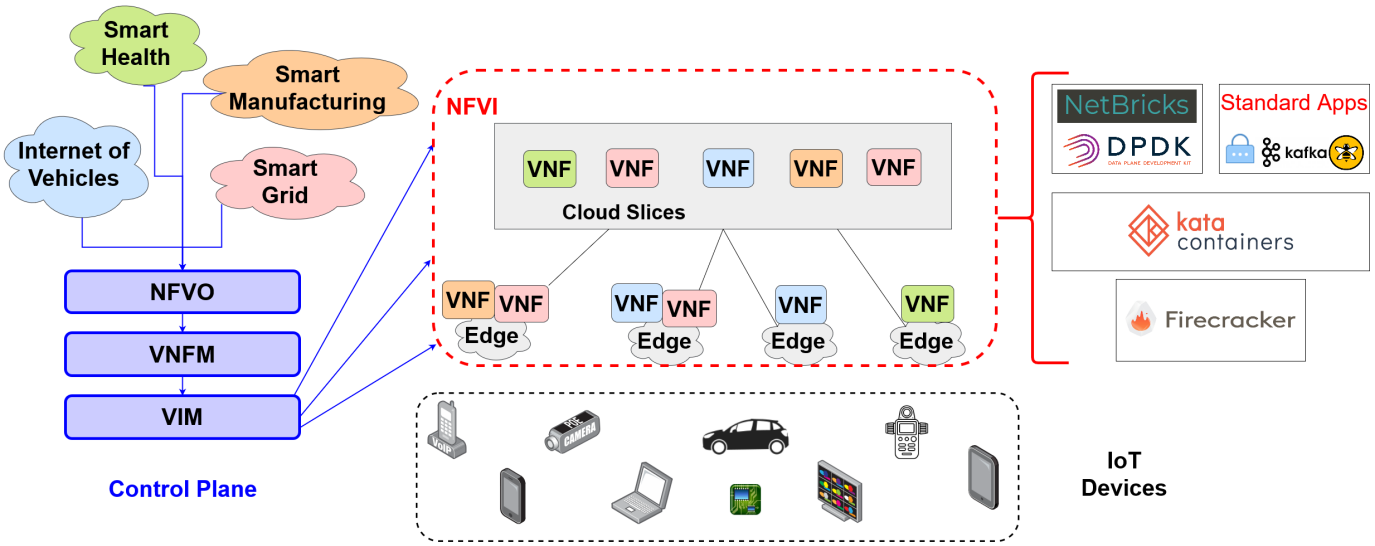

Figure 2. IoT mMTC services belonging to different scenarios can be deployed in a flexible and dynamic way by leveraging NFV features. The proposed lightweight architecture is shown as the NFV infrastructure block, compliant with the ETSI NFV standard.

The latest European Telecommunications Standard Institute (ETSI) report on the enhancement of the NFV architecture towards a cloud-native and platform-as-a-service (PaaS) approach presents some examples about the utilization of CNFs [29]. They include the use case of VNF components (VNFCs) implemented in single operating system (OS) containers, potentially wrapped in VMs. This way, they provide a nested virtualization environment and address the aforementioned security issues of CNFs. A new architecture for NFV management and orchestration (MANO), able to manage containerized VNFs, is proposed in Ref. [30] and is shown in Figure 3. It emerges that some new entities are necessary in the control plane for managing containers, as follows:

- CISM (container infrastructure service management) is the function that manages the CIS (container infrastructure service), a service offered by the NFVI, making the runtime environment available to one or more container virtualization. In turn, the CIS is exposed by one or more CIS clusters. CNFs are deployed and managed on CIS instances and make use of container cluster networks implemented in the CIS clusters [30].
- CCM (CIS cluster management) focuses on cluster management and provides lifecycle management, configuration management, performance and fault management [30].
- CIR (container image registry) manages OS container images.

These new management entities, introduced to manage CNF clusters, can be mapped onto the functions offered by Kubernetes, which is the natural implementation solution for container orchestration. The ETSI standard [30] does not specify interfaces and reference points for these new entities, thus leaving a considerable freedom in their implementation. Some documents from ETSI [31,32] or cloud providers [33] show some possibilities, together with a mapping to Kubernetes functions, but a fully agreed solution has not yet been reached. The main difficulty is due to the fact that the way ETSI NFV operates is significantly different from the Kubernetes approach. In fact, the NFV model uses the declarative approach to manage imperative operations, whereas Kubernetes uses it in an intent-based style. In more detail, NFV specifies procedures and interfaces between NFVO, VNFM, and VIM, whereas Kubernetes manages internal and external communications using artifacts, APIs, and manifest files, specifying only the intent, which is the desired final operating state.

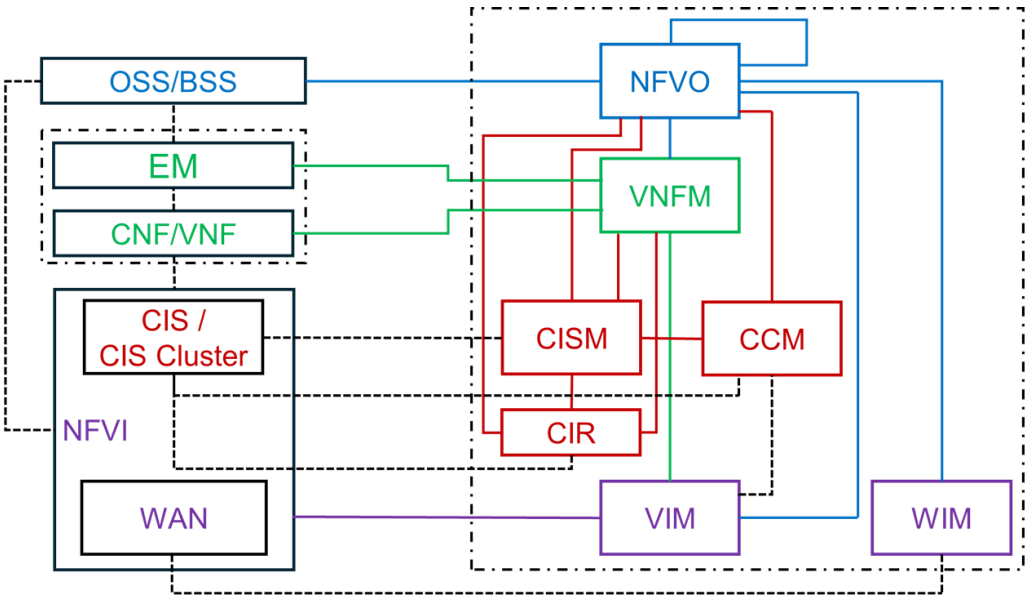

Figure 3. New entities in NFV control plane (CISM, CCM, CIR) introduced to manage CIS cluster in NFVI as envisioned in Ref. [30]. Solid lines indicate NFV interfaces, dashed lines indicate other additional interfaces. Reference points names are omitted to improve readability.

In terms of mapping, Kubernetes has an active role in both cluster provisioning (NFVI) and control (NFVO, VNFM, and VIM). In fact, the CIS function can be associated with a cluster of Kubernetes Worker nodes, capable of providing CNFs. Instead, the control counterpart (CISM) can be associated with the functions offered by the Kubernetes Master nodes. Kubernetes API implements container management service interfaces exposed by the CISM. A important role is also played by Helm, which is a tool for managing OS containers that are deployed on Kubernetes CIS clusters. The Helm CLI allows for the implementation of the OS container workload management service interface exposed by the CISM [31,32].

In terms of what concerns the CCM, it has the ability to scale the number of CIS clusters (i.e., Kubernetes clusters) based on the resources that are necessary to serve the workload offered, and is not present in the Kubernetes ecosystem; while there are dedicated proprietary tools in cloud providers such as Google, AWS or Microsoft that offer this function, porting it to the edge is more complicated. The main point is whether there are enough resources on the edge node(s) to serve the offered workload. If the answer is positive, it is necessary to introduce further tools able to provide and configure resources as additional Kubernetes clusters, either virtualized or bare metal. However, if these resources are not available locally, it is necessary to perform a bursting operation in the edge-to-cloud continuum (see, e.g., Ref. [34]), with possible performance issues. The adoption of the serverless approach should help to limit the bursting events towards remote clouds.

Regarding CIR, Kubernetes does not offer any specific service to host container images, but relies on external services. Instead, it is available as a Kubernetes Registry service, which is an image pull secret used to authenticate a private image registry, in order to pull a container image hosted on it. In this regard, it is also useful to point out the fully serverless approach allows a developer to write the code and upload it without additional operations. It will be executed in a pre-determined environment with a collection of libraries and OS installed. Thus, CIR is mainly related to the classic approach, based on containers, or for those users that, while using serverless, prefer to prepare a custom image to pull and deploy.

In general, the presence of Kubernetes also significantly changes the way in which other management entities work. In the ETSI architecture, the interaction between NFVO and VNFM implements the lifecycle operation of VNFs. In particular, VNFM requests grants from NFVO before starting operations on controlled VNFs. When Kubernetes is

used, it does not interact with NFVO to scale its controlled Pods in and out, or to decide which worker node to instantiate them on. NFVO has to define only the desired final state in a declarative style (e.g., intent-driven constructs, such as deployment and YAML files), and the Kubernetes scheduler acts to reach that state, while VNFM keeps a detailed view of the status of its controlled VNFs, including virtualization information, and exposes this information to NFVO; Kubernetes does not expose anything on its internal status. The only way to control operations is to define appropriate intents to achieve. Kubernetes manages the lifecycle of CNFs implemented in Pods and performs scaling operations through changes in deployments by specifying constraints in manifest files.

To sum up, the usage of Kubernetes breaks the tight control loop designed by ETSI MANO, replacing it with multiple small, declarative control loops, characterized by greater flexibility.

In addition, the new structure of the NFV infrastructure (NFVI), capable of supporting containerized VNFs [35] is depicted in Figure 4. It shows that the containerized VNFs have two main options: to deploy the CIS either on the bare metal or inside VMs. The first option offers more flexibility and improved performance, but lacks some security and isolation features, as discussed above. On the other side, VMs can be effectively used to provide isolation of CIS (e.g., devoted to a single slice), but they suffer from promptness, resource efficiency and performance issues.

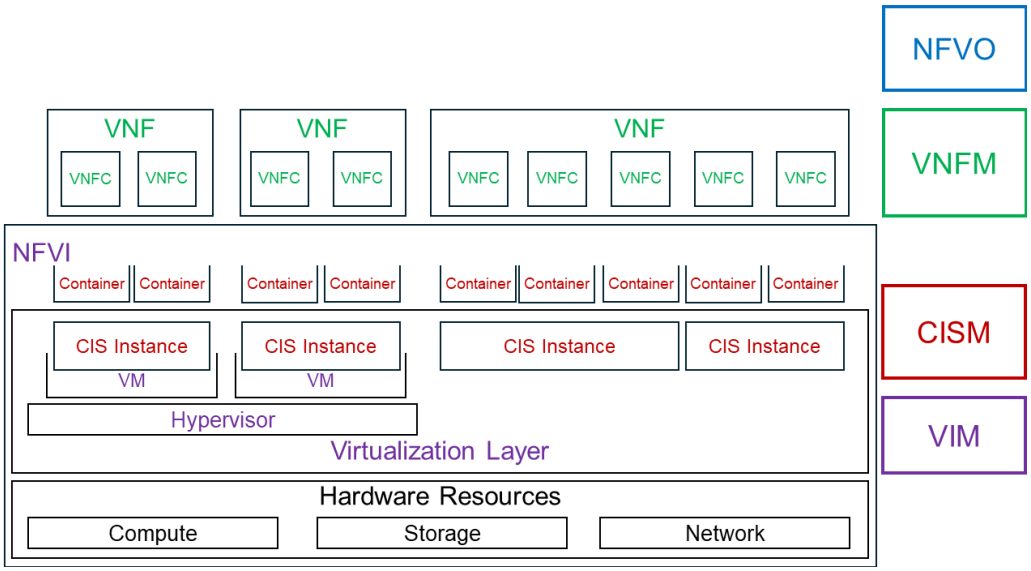

Figure 4. NFV infrastructure as envisioned in Ref. [35], providing containers on bare metal (right side of NFVI) and VMs (left side of NFVI).

Our solution addresses the limitations of the solutions envisioned in Ref. [35], which are shown in Figure 4. It is compliant with the latest ETSI NFV white paper [29], which provides a future outlook. Figure 2 shows the NFVI stack proposed in this paper. In particular, it relies on lightweight virtualization technologies in order to both overcome the limitations of traditional VMs and fulfil the CNFs' security needs. The virtualization layer consists of two sub-layers. In the top sub-layer, we use Kata containers [36], an OpenStack Foundation project. Unlike traditional container runtime technologies, such as Containerd [19], which rely on kernel features such as Cgroups and namespaces to provide isolation with the shared kernel, Kata offers better isolation by enclosing containers in lightweight VMs (*microVMs*). In particular, each Kata container runs in a dedicated and isolated kernel within a dedicated microVM. The Firecracker hypervisor [37] accelerates the deployment of Kata microVMs, provides additional isolation, and represents the lower virtualization sub-layer of our solution. New VNFCs are built and executed in NetBricks [38], an NFV framework which includes both a programming language and an execution environment. It makes use of the rust and data plane development kit (DPDK) [38]. In addition, it is also

possible to reuse standard packages, such as Kafka or MQTT brokers, which safely run in Kata containers. OpenStack or Kubernetes can be used for resource management for such infrastructure, occupying the virtual infrastructure manager (VIM) block in the NFV reference architectural framework, as shown in Figures 3 and 4.

4.1. A Network Function Virtualization Specialized Framework: NetBricks

Traditional applications frameworks, such as Hadoop and Spark, are implemented using high-level abstractions, with optimized implementation for high performance [38]. However, in the NFV context, none of these provide both high performance and rapid development, and the NFV code optimization process is usually slow. Furthermore, relying on hardware isolation between VNFs incurs significant performance overheads for simple functions. To address these issues, Ref. [38] proposed a new open-source NFV framework, called NetBricks. NetBricks includes both a programming model and an execution environment. Running network functions (NFs) in VMs or containers enforces isolation by making network I/O packets cross a hardware memory isolation boundary. This entails a context switch (or syscall), or requires that packets must cross core boundaries, with significant overhead in any case. Unikernels, as a virtualization alternative, prevent context switches by pulling application functionality into the operating system kernel, whereas NetBricks avoids this overhead by relying on compile-time and runtime checks to enforce memory isolation in software (software isolation).

Programming abstractions and zero copy soft isolation are two main components of NetBricks. In general, developers of NFs spend a lot of time meeting performance targets with their code, dealing with low level abstractions and low level code. Since NFs exhibit common patterns, the strategy in NetBricks is to abstract and optimize these patterns. Each NF in NetBricks can be defined as a directed graph, built using five basic programming abstractions as nodes, i.e., the *netbricks*, as shown with different colours in Figure 5. These nodes represent functions and operators for (i) packets processing (e.g., parsing/transforming/filtering packets based on their headers), (ii) bytestream processing (e.g., to reconstruct TCP segments), (iii) control flows (e.g., for branching and merging branches in the NF graph), (iv) state management (e.g., to relieve programmers from cross-core access of NF code on multi-core CPUs) and (v) event scheduling (e.g., to invoke specific functions beyond packet processing). The behaviour of these operators can be customized through user defined functions, provided as input arguments. The graph in Figure 5 illustrates how to define two different branches, starting from the control flow operator. One leads to more differentiated operations through another branching, while the other is used for data (bytestream) processing and also provides an abstract view of the function access to a multi-core CPU. Zero copy soft isolation represents an ensemble of techniques to ensure runtime environment safety and efficiency. The use of a safe language (Rust) and the LLVM compiler disallows pointer arithmetic and access to null objects, checking bounds on array accesses in order to prevent overflow and undefined behaviors. Moreover, the choice of Rust, which uses smart pointers for heap allocations, provides predictable latency for the applications. NFV requires that messages cannot be modified by an NF after they have been sent. For this reason, NetBricks' runtime makes use of unique types, which disallow two threads from simultaneously accessing an object, to ensure the packet is isolated without copying it during the transition from one NF to another. NetBricks works with a DPDK fast I/O library and is clean-slate, because it requires the use of its model to rewrite NFs.

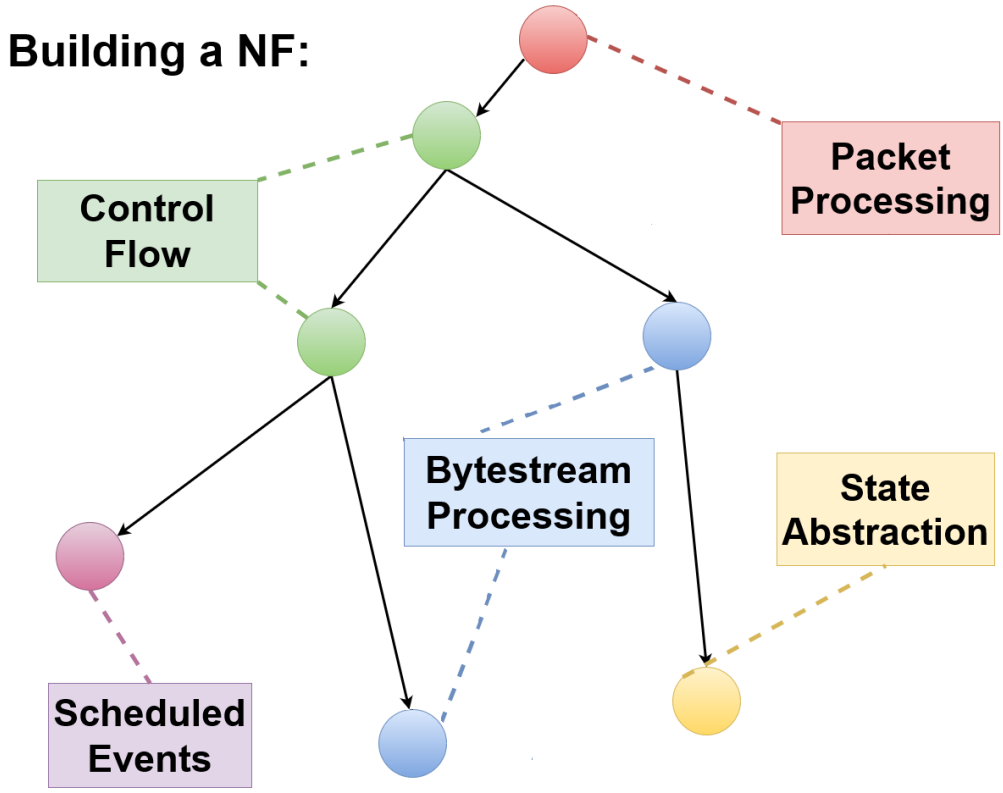

Figure 5. NetBricks defines an NF connecting blocks from five different programming abstraction.

4.2. Nesting Virtualization with Kata Containers

Kata Containers is a project started in 2017 and managed by the OpenStack Foundation. It wraps containers into dedicated microVMs and provides strong isolation with a flexibility similar to regular containers. The Kata Containers' runtime (kata-runtime) is compatible with both the Open Container Initiative (OCI) runtime specification and the Kubernetes Container Runtime Interface (CRI). Therefore, it works seamlessly with Docker Engine and the Kubernetes kubelet, deploying a KVM VM for each container or pod created, with multiple hypervisors supported. The deployment of VNFs on Kata can take advantage of some features that its microVMs expose, such as multiple interfaces, DPDK, single-root input/output virtualization (SRIOV), and MacVTap.

Figure 6 shows how Kata Containers expose Containerd and CRI to support the Kubernetes scheduling. Containerd-shim-kata-v2 acts as a mediator between Containerd and Kata containers. With its introduction, Kubernetes can launch Pod and containers with one shim per Pod instead of 2N+1 shims, with a significant efficiency improvement. Kata-agent, a daemon running within the microVM that is responsible for the spawning containers processes, communicates with containerd-shim-kata-v2 by using a gRPC server exposed via VSOCK interface. The gRPC protocol allows the runtime (integrated into shim-kata-v2) to send container management commands to the agent and carry the I/O streams between the containers and the manage engines [36]. In the NFV architecture, OpenStack can be used as a VIM to manage resources, through its Nova service. It can interoperate with Kubernetes, as already demonstrated for Baidu Container Instances in Ref. [39]. OpenStack itself, either using its Zun container service or leveraging a Kubernetes cluster, can take advantage of Kata efficiency and safety. A Kata microVM can be deployed on top of multiple hypervisors, such as QEMU, NEMU or Firecracker, specifically aiming to create and manage microVMs.

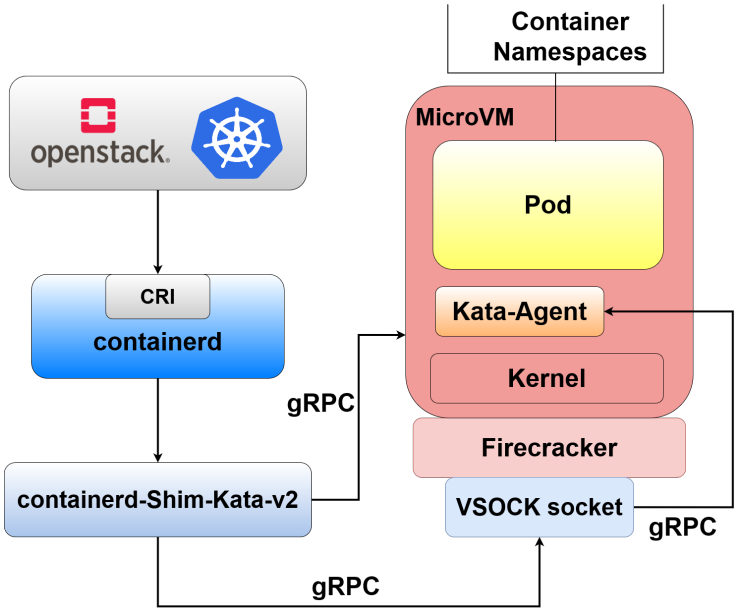

Figure 6. Kata container nests a pod, encapsulating containers' processes and namespaces, in a microVM for enhanced isolation.

4.3. A Speed Focused Virtual Machine Manager: Firecracker

Firecracker is an open-source, minimalist hypervisor written in Rust. It makes use of the Linux kernel-based virtual machine (KVM) to create and manage microVMs. Similar to Unikernels, Firecracker is built with minimal device emulation, which enables a faster startup time (150 microVMs per s [40]) and a reduced memory footprint for each microVM.

In order to configure a microVM, users can interact with Firecracker through the application programming interface (API) thread, as shown in orange in Figure 7. A Firecracker instance runs in the user space and it is isolated by a jailer, consisting of a secure computing mode (seccomp), control groups (cgroups) and namespace policies. It also exposes a serial console and a reset controller to stop microVMs. Clients can use the rate limiter, via an API, to control network and storage resources and to configure bursts or specific bandwidth/operational limitations [40]. Host data are exposed to microVMs through file block devices. It is important to consider that Firecracker's minimalist approach leads to some limitations, such as the lack of filesystem sharing with the host, device hot-plug support and dynamic resizing of resources [41], which, however, should not be critical issues in the dynamic mMTC environments.

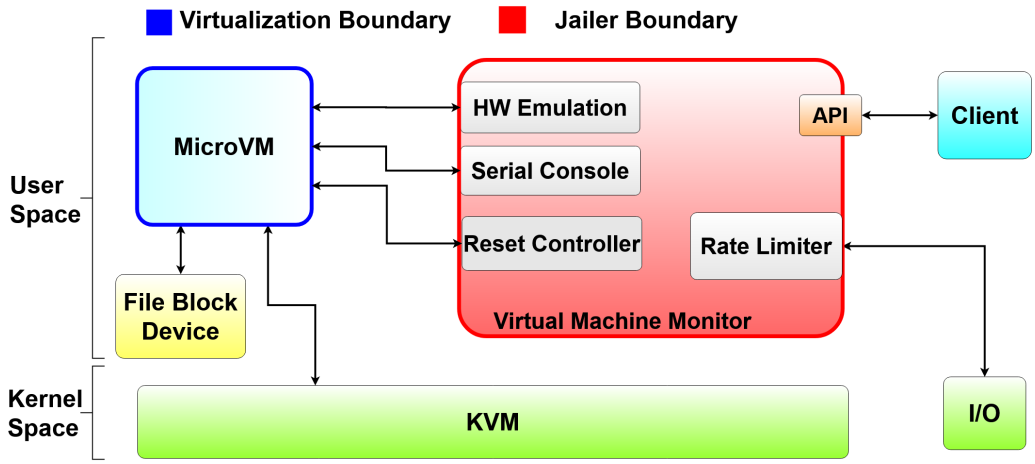

Figure 7. Firecracker enables the deployment of lightweight microVMs through a minimal virtual machine monitor.

5. Performance Evaluation

5.1. Internet of Things Virtual Network Functions Requirements

The aforementioned IoT scenarios involve the use of different types of VNFs. Table 2 reports the main features of the proposed architecture, in relation to some example VNFs.

Table 2. Impact of proposed techniques on VNF application examples in the IoT domain.

Architecture	VNF			
	Load Balancer	**C-V2X Server**	**Video Streaming Server**	**Firewall**
NetBricks	Limited memory overhead and optimized network functions	High processing rate (>25 Mbps) [38]	High processing rate (>25 Mbps) [38]	Memory and packet isolation and optimized network functions
Kata Containers	Reduced agent and VMM overhead (1 MB, 10 MB) [42]	VM-like isolation, reduced agent and VMM overhead (1 MB, 10 MB) [42]	Reduced agent and VMM overhead (1 MB, 10 MB) [42]	VM-like isolation
Firecracker	High density creation rate (150 microVM/s) [37]	Fast scaling and fast startup time (125 ms) [37]	Fast startup time (125 ms) [37]	Minimal attack surface

The load balancer is a common middlebox to be softwarized in order to improve scalability and reduce management costs. NetBricks provides optimized packet processing and forwarding functions for developing load balancers with low latency, resulting in reduced memory overhead due to Rust runtime safety checks, which can be eliminated statically in some cases (e.g., where bounds can be placed on the index statically) [38]. Moreover, the solution of Kata Containers, accelerated by the Firecracker VMM, produces fast and high density deployment, suitable to satisfy the load balancer requirements for replicability and granularity.

IoV requires services related to cooperation and communication between traffic participants, in order to optimize driving behavior for both automated systems and human drivers. To this end, wireless communication systems, such as C-V2X, can be used to exchange information about speed, position, intended trajectories/maneuvers, geolocation and other helpful data between vehicles through messages which can be periodical (CAM) or event-based (DENM) [23]. The server orchestrating these messages should be able to scale quickly when there is a high volume of traffic. To this aim, Firecracker VMM entails a rapid startup time (125 ms [37]). The stronger isolation introduced by Kata Containers provides a safer environment for the management of information messages in a security-sensitive context like IoV. Also, the extra latency introduced by the Kata nested virtualization may preclude the usage of this architecture in uRLLC services. However, there are C-V2X services, such as an alerting one [43], with timescales in the order of seconds. They are compliant with the performance of the proposed architecture in terms of both startup time of new instances and service time for data plane traffic.

Video streaming and multimedia content delivery are also commonly present in the IoV scenario [23], for both remote driving and entertainment services. The required data rate is around 2–20 Mbps [44], which can be sustained by the high processing rate of the NetBricks optimized environment [38].

Since, in IoT scenarios, security is critical, a common service is firewalling. The proposed architecture comes with a special focus on security and isolation. In addition to the isolation boundaries introduced by Kata Containers, the Firecracker minimal VMM reduces the potential attack surface. Furthermore, as mentioned above, the NetBricks framework makes use of zero copy soft isolation and Rust to provide memory and packet isolation [38].

5.2. Experimental Results

In order to precisely quantify the advantages obtainable through the proposed architecture, compared to a more classic virtualization solution, we implemented a testbed,

based on two Dell rack servers hosting the Linux kernel 5.4.0-42 and Ubuntu Server 20.04 Long Term Support. The full set of features is reported in Table 3. This hardware configuration can be considered representative of an edge computing infrastructure and is therefore suitable for carrying out our experiments. All used software is open source, without licensing costs.

Table 3. Configuration of servers used in the testbed.

Configuration	Server #1	Server #2
CPU	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40 GHz	Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30 GHz
RAM	128 GB @ 2133 MT/s	64 GB @ 2133 MT/s
Disk	280 GB (145 MB/s write speed)	280 GB (145 MB/s write speed)
Network interfaces	1 × 10 Gbps, 4 × 1 Gbps	1 × 10 Gbps, 4 × 1 Gbps
VNF packages	- *Kafka* broker [45] and *ZooKeeper* - NetBricks packet generator: *packet-test*	*Kafka* publisher, based on *Sangrenel* [46]
Virtualization software	- *Kata Containers* [36,47] as serverless container runtime with *Docker* - *Firecracker* [37,40] or *QEMU* [48] as microVM hypervisors	

First of all, in line with principle, we point out that the experimental comparison should not include the implementation of VNFs in classic always-active VMs, since this comparison would be not fair. In fact, keeping a VNF always active would clearly avoid the startup latency, at the cost of increasing the used resources significantly, but also leaves them practically unused. The procedure for scaling VMs, used to implement and replicate a given set of VNFs, up and down has a temporal timescale in the order of minutes, which is typically not compliant with intermittent services. This is what we want to avoid by using a serverless approach. As discussed in Section 2.1, in the case of serverless technologies, the startup latency of containers is also a well-known problem, which is termed a cold start [7]. Even if approaches based on AI can mitigate this issue, since they are based on statistical properties, they may either overestimate or underestimate the number of idle instances to be spawned in order to serve hypothetical future requests. In the first case, resources are over-provisioned, while in the second case, they would generate cold start events, with startup times in the order of several seconds (some experimental data are reported in Table 2 of Ref. [20], which refers to large scale measurements). In addition, containers have well-known security issues, and, to mitigate these, containers themselves are sometimes used within VMs (as in Figure 4). For these reasons, in this section, we will explore the performance of an alternative yet complementary solution, capable of limiting not only the resource footprint—and, thus, increasing the function density per node—but also reducing the start up time, thus making the cold start problem less critical. For this solution, it is important to evaluate whether its performance on traffic management capacity (measured using data plane key performance indicators, KPIs; that is, latency and packet losses) is in line with the typical requirements of IoT services. For completeness, we provide some comparative results in terms of data plane KPIs compared to either the same VNF implemented with a VM or a container in a VM, in order to show that our solution is not only comparable, but also more efficient than the state-of-the-art solutions.

To accomplish these tasks, in our tests, we carried out two experiments. The first one made use of NetBricks to implement a specific VNF, and aimed to not only quantify the memory and storage footprint of a NetBricks application, but also to evaluate the boot time of the relevant microVM. All these tests were carried out by deploying the NetBricks VNF within a Kata container, and by using QEMU [48] and Firecracker [37,40] hypervisors to evaluate differences. We stress that this is an original contribution of our work. To the best

of our knowledge, the experimental deployment of NetBricks in Kata containers is a novel solution. The second experiment focused on using a VNF implemented with a standard package, but again deployed within a Kata container. Firecracker and the classic QEMU hypervisors were used to evaluate differences. This test is of paramount importance, since it is not realistic to think of substituting all VNFs with specialized NetBricks applications, and thus it is important to see if deploying a standard VNF (in our case, a Kafka broker) in a Kata + microVM environment suffers from significant limitations, with respect to the requirements reported in Table 2.

Although these experiments were executed on single instances of sample functions and did not include integration with an orchestration platform, they worked well for our purposes. In fact, our aim was to show that the serverless functions realized with the proposed framework had minimal overheads, in terms of resource footprint. Thus, they are suitable candidates to be instantiated in many replicas in the same edge server, with a startup time compliant with mMTC requirements for the analyzed IoT use cases. In addition, we also showed that data plane KPIs of a single instance are compliant with most service requirements, also beyond those of plain mMTC services.

5.2.1. NetBricks Virtual Network Function Experiment

The experimental comparison was organized as follows. An image of Ubuntu 17.10 hosting NetBricks was deployed with a Kata container (v.1.11.2) using both QEMU, which was the baseline solution, and Firecracker v.0.21.1. These two hypervisors are currently supported by Kata. The widespread QEMU was expected to guarantee an extensive hardware support, while Firecracker brought a technology that minimized resource footprint and workloads. The first experimental observation was relevant to the disk footprint of the two hypervisors. As expected, they required a significantly different storage. In particular, QEMU required 16 MB of disk space, whereas Firecracker required just 2.6 MB. This was significant since, in situations of limited availability of resources, as it should be in MEC nodes or resource-limited devices, a saving of disk space of more than 80% is a huge benefit. If a large number of VNFs (from hundreds to thousands) has to run concurrently on the same node, the storage requirement of Firecracker would be almost negligible, whereas that of QEMU could become significant.

More insights can be obtained by executing VNFs in the proposed architecture. Figure 8 shows RAM usage related to the execution of NetBricks on a Kata container, with QEMU and Firecracker as a function of time. Both of them were executed in the following two different scenarios: deployment (labeled as "Hypervisor" in Figure 8) and deployment with subsequent VNF invocation (labeled as "VNF"). In this example, the VNF realized with NetBricks is a packet generator included in the framework, *packet-test*. We present the performance achieved by this VNF to show the impact of a single VNF on the resource footprint.

We can observe that, within a second, from a starting point of 559 MB of RAM already allocated on the server by other processes, deploying a NetBricks container (dotted line) Firecracker resulted in an overhead of 3 MB (red curve) while the QEMU was 20 MB (blue curves). Furthermore, in the container deployment with the execution of the packet generator VNF, Firecracker was lighter than QEMU, with a difference of about 10 MB over a total of about 41 MB; that is, a saving in memory footprint of about 25%. The vertical lines in Figure 8 show the average boot system + user time; Firecracker deployed NetBricks in $T_{boot,F}$ =71 ms and QEMU in $T_{boot,Q}$ =83 ms, respectively. This implies a 14% saving in boot time, which was significant. From our experiments, it emerged that the resource footprint and deployment latency values were similar to those characterizing most of the VNFs made available in NetBricks. Thus, the presented use case is representative of real scenarios, such as those illustrated in Section 3. From the experimental results we can identify the most suitable deployment strategies for VNFs in future IoT applications. Essentially, the selected strategies depend on the latency requirements of applications. For example, uRLLC applications characterizing the IoV and some smart manufacturing applications are

highly sensitive to latency values. Even in some smart health applications, such as remote surgery, the VNF deployment time should be larger than the latency tolerance. For all other applications, including entertainment and traffic management services in IoV, remote patient monitoring in smart health applications, or smart grid surveillance, the dynamic behavior of the proposed solution can significantly reduce the resource footprint and maximize the usage of computing and storage resources, both in MEC-based applications and in the core of a cloud network. In fact, the boot time of a VNF is by far inferior to the tolerated latency; thus, it is possible to start up a new VNF instance only when strictly necessary, and possible prediction errors, due to the AI-based scheduling system, can be tolerated.

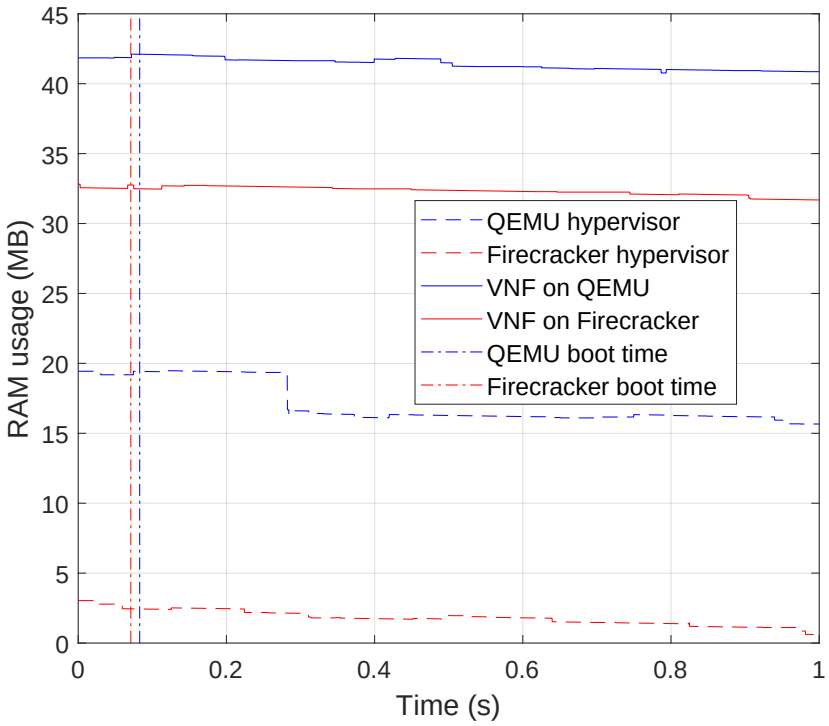

Figure 8. RAM usage and boot time during NetBricks Kata container deployment using QEMU and Firecracker hypervisors. Red lines refer to Firecracker RAM usage, and blue lines refer to QEMU RAM usage. Dotted lines indicate memory consumption for simple container deployment, whereas continuous lines represent RAM usage overhead required by container deployment followed by a VNF call. The two dash–dotted vertical lines show the average container boot time for Firecracker (red) and QEMU (blue).

5.2.2. Kafka Broker Virtual Network Function Experiment

The second experiment makes use of both servers. One of them is used to generate traffic by means of the Sangrenel tool [46], which is a well-known Kafka client. The other one is used to receive and process it using a Kafka broker [45], implemented in the VNF under test. The workload produced by Sangrenel consists of a maximum of 20 topics, with a resulting traffic rate approximately equal to 102.4 Mbps. On average, the traffic rate associated with each topic is 5.12 Mbps. The Kafka message size is 800 B. We used constant rate traffic because our goal was to evaluate the peak rate that is sustainable for a long time using our solution with the test application (worst case). This allows for the provision of additional insights on the service capacity of the implemented VNF. In our experiments, the duration of each test was 5 min. For this application, we considered the following data plane KPIs:

- *I/O message rate*: The handled throughput normalizes to the publication rate. Specifically, it represents the fraction of traffic that is correctly handled by a VNF. A Kafka

message is considered undelivered if the broker does not return the corresponding ACK message back to the Sangrenel client by the timeout.

- *CPU consumption*: the percentage of a CPU core consumed by the VNF and measured on Server #1 using a CPU collector based on the *mpstat* command.
- *Batch write latency* or simply *latency*: The time spent by the client while waiting for an ACK message from the Kafka broker. Since the Kafka client and broker run in two servers, located in the same rack and belonging to the same 10G LAN, the message propagation time is negligible. It is thus possible to estimate the service response time. In order to minimize latency, in our experiments, the batch size was set as equal to 1 Kafka message.

The first performance we analyzed was the I/O message rate, since it allowed us to evaluate the maximum number of topics that could be handled by our VNF running in a Kata container over a QEMU/Firecracker microVM. Figure 9 shows the value of the I/O message rate as a function of the number of Kafka topics used, along with the relevant 95% confidence intervals, which appeared to be almost negligible. The first comment is that the two solutions—that is, Kata + QEMU and Kata + Firecracker—performed in a similar way. In fact, although the blue line, representing Kata + QEMU, was steadily higher than the Kata + Firecracker curve, this happened for values that were not acceptable; that is, for more than five Kafka topics. For instance, for 10 topics, corresponding to a traffic rate of about 51.2 Mbps, the message loss for Kata + QEMU was equal to 3.5%, which was definitely excessive. Instead, up to five topics, corresponding to a rate equal to 25.6 Mbps, which is in line with most of the applications considered in previous sections, the message loss for Kata + QEMU was 0.17%. This was very close to that of Kata + Firecracker, which was equal to 0.41%. Both results were acceptable. It is worth noting that, for one topic, corresponding to 5.12 Mbps, the I/O message rate was equal to 100% of delivered messages. Finally, it is important to point out that this application needs to be written on a disk, which is critical for this virtualization environment. In fact, the storage was not implemented with SSD devices, which is an intrinsic limitation of Firecracker regarding access to storage. It would clearly perform better for other applications. Thus, this can be regarded as a lower bound of the message rate.

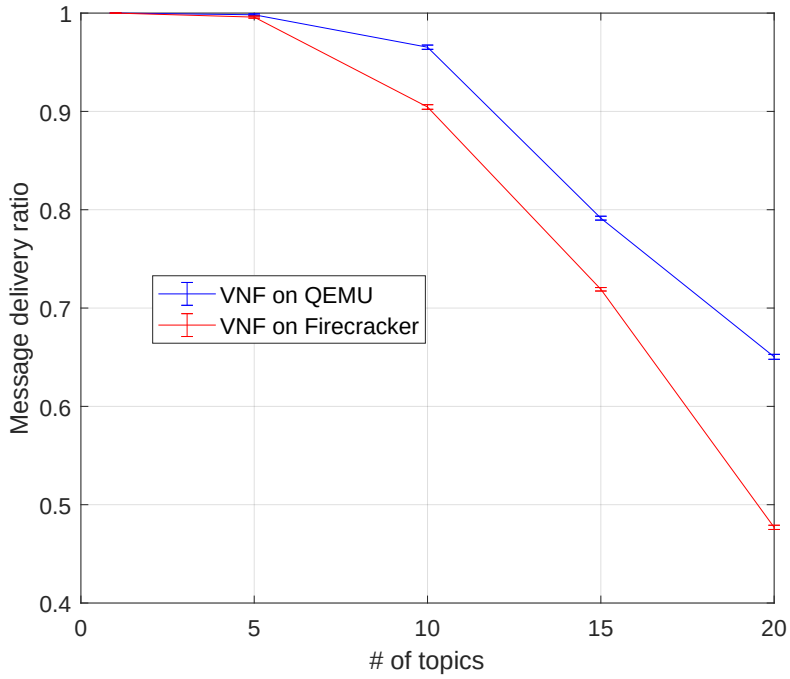

Figure 9. Message delivery ratio of a Kafka broker VNF running in a Kata container on a (blue) QEMU hypervisor, and a (red) Firecracker hypervisor.

The second KPI we analyzed was the CPU consumption, shown in Figure 10. Again, it emerged that the two frameworks performed very similarly to each other up to five topics, and then the Kata + QEMU solution consumed more CPU time. This is easily explainable, since it was also able to handle a slightly larger amount of Kafka traffic. The general comment was that such a CPU consumption is very low (less than 4.5% of a CPU core for handling a traffic rate of about 25 Mbps). Thus, again, the resource footprint of the proposed solution was more affordable than in edge/MEC nodes.

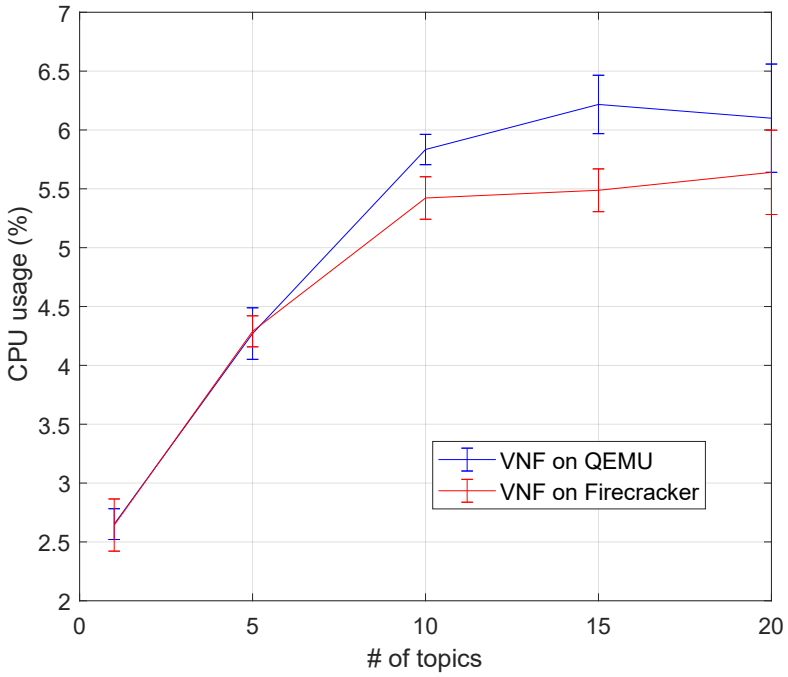

Figure 10. Percentage CPU consumption of a Kafka broker VNF running in a Kata container on a (blue) QEMU hypervisor, and a (red) Firecracker hypervisor.

Regarding the concerns of the final KPI—that is, the latency in handling Kafka messages— the two solutions looked, again, very similar. The latency values were approximately 3 ms for handling a traffic rate of 25 Mbps. This value was low enough to support practically all the use cases surveyed in previous sections. Since the deployed VNF could be modeled in the user plane as a single server deterministic model with a constant offered traffic rate (D/D/1 model in Kendall's notation), it was possible to evaluate the input rate at which the system was no longer stable and started losing messages. This rate value could be identified as the one corresponding to five topics. In fact, when five topics were used, the message loss was nearly negligible; thus, the service arrival rate λ_p could be considered equal to the Kafka broker service rate μ_p in saturation. The estimated rate was 4 Kpps, being the message size equal to $L_p = 800$ B. Thus, the service capacity C was equal to $C = L_p\mu_p$. Once this value was known, it was possible to estimate the average service time in more general conditions, using the general formulation provided by the G/G/1 model. The use of a single server model was motivated by the fact that the service consisted of the execution of work by all components of the IoT service (Kafka broker) to satisfy each user request. For this reason, the service process statistics were indicated as general in Kendall's notation. Regarding the average service rate, given the average packet length $\mathbb{E}[L]$, then $\mu = \frac{C}{\mathbb{E}[L]}$. If the statistics on service and arrival processes are known, it is possible to use Kingman's formula to approximate the mean waiting time T_w [49], as follows:

$$T_w \approx \left(\frac{\rho}{1-\rho}\right)\left(\frac{c_a^2 + c_s^2}{2}\right)\frac{1}{\mu}, \quad (1)$$

where $\rho = \frac{\lambda}{\mu}$ is referred to as average utilization, whereas c_a and c_s are the coefficients of variation of arrival times and service times, respectively. This means that $c_a = \sigma_a \lambda$, with σ_a being the standard deviation of arrival times, and $c_s = \sigma_s \mu$, where σ_s is the standard deviation of arrival times. From the equation above, it follows that the overall mean service time T_s can be estimated as follows:

$$T_s \approx T_w + \frac{1}{\mu}. \tag{2}$$

When we move away from deterministic arrival or services processes, the condition of $\rho \approx 1$ cannot be clearly approached, as the system becomes unstable.

The Kingman's approximation is known to work well in situations with heavy load that are, thus, close to saturation. This is indeed a possible situation in edge computing scenarios, where the available resources are limited. For lighter load conditions, a good approximation is given by the M/G/1 model, from which the Pollaczek–Khinchine formula [50] is derived. It assumes a Poisson arrival process, and provides a useful relationship between the mean waiting time T_w and the service rate μ, as follows:

$$T_w = \frac{\lambda\left(\sigma_s^2 + (1/\mu)^2\right)}{2(1-\rho)}, \tag{3}$$

where σ_s^2 is the variance of the service time distribution. Since the Poisson approximation is acceptable for the arrival process when many low-rate and non-deterministic traffic sources combine together, this model can also be useful for predicting the response time of a VNF, once the distribution of the service time is known.

However, when considering a real edge deployment, it is not enough to estimate the service time from a VNF or a cascade of them, but it is important to also take into account the latency to and from the edge location. In Ref. [51], the authors estimated the average downlink and uplink for a 4G network through a measurement campaign. It resulted in an average uplink latency of $T_{uplink} = 28.84$ ms, with a standard deviation of $\sigma_{uplink} = 18.64$ ms, an average value on the downlink side equal to $T_{downlink} = 18.63$ ms, and a standard deviation of $\sigma_{downlink} = 6.39$ ms. Therefore, the average round trip time was approximately 50 ms, which should be taken into account when compiling the latency budget, including any VNF startup time. All these values are definitely higher than the service time that is presented in Figure 11, which is lower than 10 ms. The situation becomes even trickier when vehicular devices are considered, as in C-V2X services. In this case, it is critically important to take into account not only the service time, any startup time and round-trip latency, but also the size of the service area and the travel speed. In fact, if a VNF must be transferred to another edge/MEC node due to the vehicle crossing the service area border, it is necessary to carefully evaluate the possible advance with which this operation has to be performed. If the service tolerates sub-second delays, like standard notification services that use CAM messages, it may be possible to avoid any early instantiation. In fact, from the moment a message is sent once the border of the service area is crossed, the overall time to load the information on the border would be equal to $T_{up,max} = T_{uplink} + T_{RNIS} + T_{boot,F} + T_s$, where T_{RNIS} is the time needed by the Radio Network Information Service (RNIS) to detect the change in service area and to trigger the VNF instantiation. T_{RNIS} is estimated in the order of few ms and is assumed to be equal to approximately 4–5 ms [52]. Using the values mentioned above, we can estimate a $T_{up,max}$ value of about 100–110 ms. Considering a vehicle speed of 120 km/h, typical of highways, we obtain the fact that, when the information was processed by the on-board C-V2X service, the vehicle could have moved less than 4 m. If the vehicle needs a response back, the overall round trip time has to also include $T_{downlink}$, leading to about 120–130 ms, which corresponds to a vehicle movement of less than 4.5 m in the meantime. This value is acceptable for most services, with the exception of those that manage automatic maneuvers, which instead require uRLLC services with bounded times.

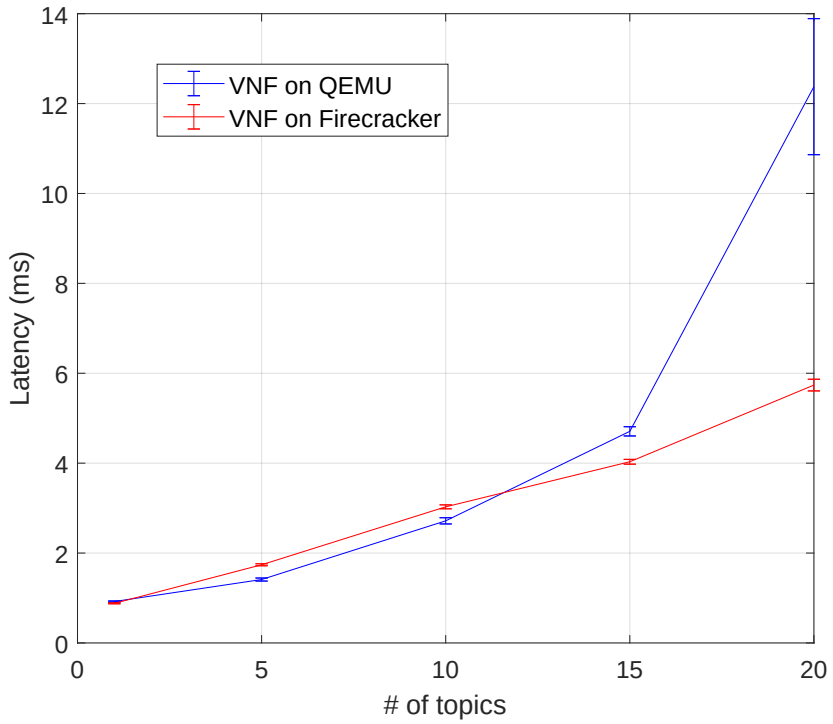

Figure 11. Batch write latency (in ms) for a Kafka broker VNF running in a Kata container on (blue) a QEMU hypervisor, and (red) a Firecracker hypervisor.

Finally, we provide some comparative results, in terms of data plane KPIs, compared to the Kafka VNF broker implemented with VMs or containers in VMs. In order to compare VNFs that have quite different deployment models, we consider a normalized performance metric, defined as the CPU occupancy required by a Kafka VNF to handle 1 Mbps of traffic. Experimental results are shown in Table 4 for the following three deployments:

- Kafka broker VNF, running inside a VM implemented with Ubuntu OS;
- Kafka broker VNF, running in a Docker container inside the same VM;
- Kafka broker VNF, running in a Kata container inside a Firecracker microVM.

We emphasize that these three deployments all provide the same level of security and isolation. For this reason, given the isolation issues affecting containers, the test with the Kafka broker VNF running in a Docker container on the bare metal was not considered.

Table 4. Comparison of the ability of different deployment models to handle 1Mbps of Kafka traffic.

Deployment Model for Kafka Broker VNF	Amount of CPU Power
VM	0.045
Docker in a VM	0.054
Kata + Firecracker	0.034

The results are reported in Table 4. It is clear that our approach is not only the fastest at starting a new VNF, making it a great candidate for adoption in a serverless approach, but it is also more efficient than the baseline competitors. In fact, in order to handle the same volume of traffic, it is the one that requires the least amount of CPU capacity. Clearly, using Docker containers within a VM involves additional overhead, which is balanced, in operation, by the ability to flexibly allocate the VM's computing power to different functions implemented with different containers. However, isolation is only granted at the VM boundary. Our approach maintains the same flexibility, but also guarantees isolation at the microVM level, thus presenting itself as a good candidate for the implementation of

CNF, as in the right side of Figure 4. From a numerical viewpoint, the results show that our VNF requires about 0.86 CPU occupancy to handle 25 Mbps, whereas the other approaches would require more than 1 CPU core occupancy.

To sum up, it is possible to say that the solution of Kata + Firecracker has a lower resource footprint than the alternative deployment solution based on a classic QEMU hypervisor, in terms of storage and memory. They have similar CPU consumption levels, which essentially depend on the VNF and not on the underlying virtualization solution. In addition, data plane performance is fully aligned to the baseline solution, based on QEMU, and it is capable of satisfying the requirement of most IoT applications, with a significant speed increase in terms of boot time, which is a major bonus.

6. Conclusions

This paper shows a lightweight architecture for implementing network services through NFV. It exploits the flexibility of the serverless approach to deploy evolved IoT applications, such as IoV, smart manufacturing, smart health and smart grid applications.

This proposed architecture includes innovative technologies, such as NetBricks, which is an NFV framework focused on a safe runtime and optimized function modules. It also includes Kata Containers, a lightweight container runtime which makes use of a microVM to wrap applications and improve isolation, which can be boosted by Firecracker. The latter is a VMM based on KVM, with fast startup times and a low memory footprint. The combination of low resource usage and robust isolation also makes this architecture suitable for edge/MEC-based service platforms, which are typically used for implementing IoT applications. Security is also improved due to the reduction in the attack surface of microVMs instantiated by Firecracker.

The experimental results show that the proposed architecture exhibits a significant reduction in both resource footprint, in comparison with legacy approaches, and VNF instantiation time, which may be a critical parameter for serverless computing. Data plane performance fully satisfies the performance requirements of most IoT applications, and are aligned with those based on the classic QEMU hypervisor. However, a trade-off between latency and resource footprint, along with the introduction of extra layers of isolation, exists. This trade-off makes the proposed system unsuitable for some very demanding uRLLC applications, and this is actually still an open area of research.

In our system, we used both NetBricks and standard packages, both encapsulated in Kata containers, running in Firecrackers microVMs. Although NetBricks cannot be a suitable choice for all applications, thanks to its efficiency in packet processing, which is accelerated by the usage of DPDK, it can be an excellent solution for implementing low level functions needing fast packet processing, such as VPN gateways. However, in complex services made of several functions, such as vehicular or video streaming services, some high-level functions can be implemented using already available packages, running in Kata containers and hosted in Firecracker microVMs. We have shown that, even in this case, our proposal provides significant advantages over the baseline solutions.

The proposed solution is not an alternative to using AI techniques to anticipate the instantiation of VNFs in a serverless framework, but rather is complementary to them, since it is able to minimize the impact of an incorrect decision.

Future works will consider a larger experimental campaign, using Kubernetes to perform resource management and orchestration. This would also include mobility services, to evaluate with real tests the time advance needed to generate a VNF on a different edge node to guarantee service continuity, depending on the service requirements.

Author Contributions: Conceptualization, G.R.; methodology, G.R.; validation, M.F.; writing—original draft preparation, M.F.; writing—review and editing, G.R.; funding acquisition, M.F. and G.R. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the European Union—NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041—

VITALITY, and the MUR Extended Partnerships grant PE00000001—RESTART. We acknowledge Università degli Studi di Perugia and MUR for support within the projects VITALITY and RESTART.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

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

Abbreviations

The following abbreviations are used in this manuscript:

3GPP	3rd Generation Partnership Program
API	Application Programming Interface
C-V2X	Cellular Vehicle-to-Everything
CCM	CIS Cluster Management
CIR	Container Image Registry
CIS	Container Infrastructure Service
CISM	Container Infrastructure Service Management
CNF	Cloud-Native Network Function
CPU	Central Processing Unit
CRI	Container Runtime Interface
DPDK	Data Plane Development Kit
E2E	End-to-End
eMBB	Enhanced Mobile BroadBand
eMTC	Enhanced Machine-Type Communications
FaaS	Function-as-a-Service
gRPC	General-Purpose Remote Procedure Call
IoT	Internet of Things
KVM	Kernel-Based Virtual Machine
MANO	Management and Orchestration
MEC	Multi-Access Edge Computing
mMTC	Massive Machine-Type Communications
NF	Network Function
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NFVM	Virtual Network Function Manager
NFVO	Network Function Virtualization Orchestrator
OCI	Open Container Initiative
QoS	Quality of Service
RAM	Random Access Memory
uRLLC	Ultra-Reliable Low Latency Communications
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VMM	Virtual Machine Manager
VNF	Virtual Network Function
VNFC	Virtual Network Function Component
VSOCK	VM Sockets

References

1. Vaezi, M.; Azari, A.; Khosravirad, S.R.; Shirvanimoghaddam, M.; Azari, M.M.; Chasaki, D.; Popovski, P. Cellular, Wide-Area, and Non-Terrestrial IoT: A Survey on 5G Advances and the Road Toward 6G. *IEEE Commun. Surv. Tutor.* **2022**, *24*, 1117–1174. [CrossRef]
2. Veedu, S.N.K.; Mozaffari, M.; Hoglund, A.; Yavuz, E.A.; Tirronen, T.; Bergman, J.; Wang, Y.P.E. Toward Smaller and Lower-Cost 5G Devices with Longer Battery Life: An Overview of 3GPP Release 17 RedCap. *IEEE Commun. Stand. Mag.* **2022**, *6*, 84–90. [CrossRef]
3. Wang, C.X.; You, X.; Gao, X.; Zhu, X.; Li, Z.; Zhang, C.; Wang, H.; Huang, Y.; Chen, Y.; Haas, H.; et al. On the Road to 6G: Visions, Requirements, Key Technologies, and Testbeds. *IEEE Commun. Surv. Tutor.* **2023**, *25*, 905–974. [CrossRef]
4. Mijumbi, R.; Serrat, J.; Gorricho, J.L.; Bouten, N.; De Turck, F.; Boutaba, R. Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Commun. Surv. Tutor.* **2016**, *18*, 236–262. [CrossRef]

5. Aditya, P.; Akkus, I.E.; Beck, A.; Chen, R.; Hilt, V.; Rimac, I.; Satzke, K.; Stein, M. Will Serverless Computing Revolutionize NFV? *Proc. IEEE* **2019**, *107*, 667–678. [CrossRef]
6. Milojicic, D. The Edge-to-Cloud Continuum. *Computer* **2020**, *53*, 16–25. [CrossRef]
7. Wang, L.; Li, M.; Zhang, Y.; Ristenpart, T.; Swift, M. Peeking Behind the Curtains of Serverless Platforms. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18), Boston, MA, USA, 11–13 July 2018; pp. 133–146.
8. Raith, P.; Nastic, S.; Dustdar, S. Serverless Edge Computing—Where We Are and What Lies Ahead. *IEEE Internet Comput.* **2023**, *27*, 50–64. [CrossRef]
9. Benedetti, P.; Femminella, M.; Reali, G.; Steenhaut, K. Reinforcement Learning Applicability for Resource-Based Auto-scaling in Serverless Edge Applications. In Proceedings of the 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops), Pisa, Italy, 21–15 March 2022; pp. 674–679. [CrossRef]
10. Cassel, G.A.S.; Rodrigues, V.F.; da Rosa Righi, R.; Bez, M.R.; Nepomuceno, A.C.; André da Costa, C. Serverless computing for Internet of Things: A systematic literature review. *Future Gener. Comput. Syst.* **2022**, *128*, 299–316. [CrossRef]
11. Wang, I.; Liri, E.; Ramakrishnan, K.K. Supporting IoT Applications with Serverless Edge Clouds. In Proceedings of the 2020 IEEE 9th International Conference on Cloud Networking (CloudNet), Virtual Conference, 9–11 November 2020; pp. 1–4. [CrossRef]
12. Benedetti, P.; Femminella, M.; Reali, G.; Steenhaut, K. Experimental Analysis of the Application of Serverless Computing to IoT Platforms. *Sensors* **2021**, *21*, 928. [CrossRef]
13. Djemame, K.; Parker, M.; Datsev, D. Open-source Serverless Architectures: an Evaluation of Apache OpenWhisk. In Proceedings of the 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), Leicester, UK, 7–10 December 2020; pp. 329–335. [CrossRef]
14. Persson, P.; Angelsmark, O. Kappa: serverless IoT deployment. In Proceedings of the 2nd International Workshop on Serverless Computing, Las Vegas, NV, USA, 11–15 December 2017; pp. 16–21. [CrossRef]
15. López Escobar, J.J.; Díaz-Redondo, R.P.; Gil-Castiñeira, F. Unleashing the power of decentralized serverless IoT dataflow architecture for the Cloud-to-Edge Continuum: a performance comparison. *Ann. Telecommun.* **2024**. [CrossRef]
16. Mistry, C.; Stelea, B.; Kumar, V.; Pasquier, T. Demonstrating the Practicality of Unikernels to Build a Serverless Platform at the Edge. In Proceedings of the 2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Bangkok, Thailand, 14–17 December 2020; pp. 25–32. [CrossRef]
17. Pinto, D.; Dias, J.; Ferreira, H.S. Dynamic Allocation of Serverless Functions in IoT Environments. In Proceedings of the 2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC), Los Alamitos, CA, USA, 29–31 October 2018; pp. 1–8. [CrossRef]
18. Ferry, N.; Dautov, R.; Song, H. Towards a Model-Based Serverless Platform for the Cloud-Edge-IoT Continuum. In Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 16–19 May 2022; pp. 851–858. [CrossRef]
19. Containerd—An Industry-Standard Container Runtime with an Emphasis on Simplicity, Robustness and Portability. Available online: https://containerd.io/ (accessed on 2 March 2024).
20. Mahmoudi, N.; Khazaei, H. Performance Modeling of Serverless Computing Platforms. *IEEE Trans. Cloud Comput.* **2022**, *10*, 2834–2847. [CrossRef]
21. Sultan, S.; Ahmad, I.; Dimitriou, T. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access* **2019**, *7*, 52976–52996. [CrossRef]
22. Huawei Technologies. *5G Unlocks a World of Opportunities: Top Ten 5G Use Cases*; Huawei Technologies: Shenzhen, China, 2017. Available online: https://www-file.huawei.com/-/media/corporate/pdf/mbb/5g-unlocks-a-world-of-opportunities-v5.pdf?la=en (accessed on 14 July 2020).
23. Kumar, D.; Rammohan, A. Revolutionizing Intelligent Transportation Systems with Cellular Vehicle-to-Everything (C-V2X) technology: Current trends, use cases, emerging technologies, standardization bodies, industry analytics and future directions. *Veh. Commun.* **2023**, *43*, 100638. [CrossRef]
24. Thalanany, S.; Hedman, P. *Description of Network Slicing Concept*; NGMN Alliance: Düsseldorf, Germany, 2016.
25. Zhang, S. An Overview of Network Slicing for 5G. *IEEE Wirel. Commun.* **2019**, *26*, 111–117. [CrossRef]
26. Ahmed, T.; Alleg, A.; Marie-Magdelaine, N. An Architecture Framework for Virtualization of IoT Network. In Proceedings of the IEEE Conference on Network Softwarization NetSoft, Paris, France, 24–28 June 2019.
27. Zhang, Y.; Crowcroft, J.; Li, D.; Zhang, C.; Li, H.; Wang, Y.; Yu, K.; Xiong, Y.; Chen, G. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, Boston, MA, USA, 11–13 July 2018; pp. 173–186.
28. Talbot, J.; Pikula, P.; Sweetmore, C.; Rowe, S.; Hindy, H.; Tachtatzis, C.; Atkinson, R.; Bellekens, X. A Security Perspective on Unikernels. In Proceedings of the 2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), Dublin, Ireland, 15–19 June 2020; pp. 1–7. [CrossRef]
29. Cai, X.; Deng, H.; Lingli Deng, A.E.; Gao, S.; Nicolas, A.M.D.; Nakajima, Y.; Pieczerak, J.; Triay, J.; Wang, X.; Xie, B.; et al. *Evolving NFV towards the Next Decade*; ETSI White Paper No. 54; ETSI: Sophia Antipolis, France, 2023.
30. *ETSI GS NFV 006 V4.4.1 (2022-12)*; Network Functions Virtualisation (NFV) Release 4; Management and Orchestration; Architectural Framework Specification. European Telecommunications Standards Institute (ETSI): Sophia Antipolis, France, 2022.

31. *ETSI GS NFV-SOL 018 V4.3.1*; Network Functions Virtualisation (NFV) Release 4; Protocols and Data Models; Profiling Specification of Protocol and Data Model Solutions for OS Container Management and Orchestration. European Telecommunications Standards Institute (ETSI): Sophia Antipolis, France, 2022.
32. *ETSI GS NFV-IFA 040 V4.2.1*; Network Functions Virtualisation (NFV) Release 4; Management and Orchestration; Requirements for Service Interfaces and Object Model for OS Container Management and Orchestration Specification. European Telecommunications Standards Institute (ETSI): Sophia Antipolis, France, 2021.
33. AWS Whitepaper. *ETSI NFVO Compliant Orchestration in the Kubernetes/Cloud Native World*; AWS: Seattle, WA, USA, 2022.
34. Femminella, M.; Palmucci, M.; Reali, G.; Rengo, M. Attribute-Based Management of Secure Kubernetes Cloud Bursting. *IEEE Open J. Commun. Soc.* **2024**, *5*, 1276–1298. [CrossRef]
35. *ETSI GR NFV-IFA 029 V3.3.1*; Network Functions Virtualisation (NFV) Release 3; Architecture; Report on the Enhancements of the NFV Architecture Towards "Cloud-Native" and "PaaS". European Telecommunications Standards Institute (ETSI): Sophia Antipolis, France, 2019.
36. Kata Containers Architecture. 2019. Available online: https://github.com/kata-containers/documentation/blob/master/design/architecture.md (accessed on 31 January 2024).
37. Agache, A.; Brooker, M.; Iordache, A.; Liguori, A.; Neugebauer, R.; Piwonka, P.; Popa, D.M. Firecracker: Lightweight Virtualization for Serverless Applications. In Proceedings of the USENIX NSDI 20, Santa Clara, CA, USA, 25–27 February 2020; pp. 419–434.
38. Panda, A.; Han, S.; Jang, K.; Walls, M.; Ratsanamy, S.; Shenker, S. NetBricks: Taking the V out of NFV. In Proceedings of the USENIX NSDI 16, Santa Clara, CA, USA, 16–18 March 2016.
39. Yu, Z. The Application of Kata Containers in Baidu AI Cloud. 2019. Available online: http://katacontainers.io/baidu (accessed on 31 January 2024).
40. Firecracker Design. 2018. Available online: https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md (accessed on 22 January 2020).
41. OpenStack Foundation. Open Collaboration Evolving the container landscape with Kata Containers and Firecracker. In Proceedings of the Open Infrastructure Summit 2019, Denver, CO, USA, 29 April–1 May 2019.
42. Wang, X. *Kata Containers: Virtualization for Cloud-Native*; Medium: San Francisco, CA, USA, 2019. Available online: https://medium.com/kata-containers/kata-containers-virtualization-for-cloud-native-f7b11ead951 (accessed on 31 January 2024).
43. Halili, R.; Yousaf, F.Z.; Slamnik-Krijestorac, N.; Yilma, G.M.; Liebsch, M.; Berkvens, R.; Weyn, M. Self-Correcting Algorithm for Estimated Time of Arrival of Emergency Responders on the Highway. *IEEE Trans. Veh. Technol.* **2023**, *72*, 340–356. [CrossRef]
44. Yu, Y.; Lee, S. Remote Driving Control With Real-Time Video Streaming Over Wireless Networks: Design and Evaluation. *IEEE Access* **2022**, *10*, 64920–64932. [CrossRef]
45. Apache Kafka. Available online: https://kafka.apache.org/ (accessed on 31 January 2024).
46. Alquiza, J. Sangrenel. 2020. Available online: https://github.com/jamiealquiza/sangrenel (accessed on 31 January 2024).
47. Kata Containers—The Speed of Containers, The Security of VMs. Available online: https://katacontainers.io (accessed on 31 January 2024).
48. QEMU—A Generic and Open Source Machine Emulator and Virtualizer. Available online: https://www.qemu.org/ (accessed on 31 January 2024).
49. Kingman, J.F.C. The single server queue in heavy traffic. *Math. Proc. Camb. Philos. Soc.* **1961**, *57*, 902–904. [CrossRef]
50. Kingman, J.F.C. The first Erlang century—And the next. *Queueing Syst.* **2009**, *63*, 3–12. [CrossRef]
51. Slamnik-Kriještorac, N.; Yousaf, F.Z.; Yilma, G.M.; Halili, R.; Liebsch, M.; Marquez-Barja, J.M. Edge-Aware Cloud-Native Service for Enhancing Back Situation Awareness in 5G-Based Vehicular Systems. *IEEE Trans. Veh. Technol.* **2024**, *73*, 660–677. [CrossRef]
52. Coronado, E.; Raviglione, F.; Malinverno, M.; Casetti, C.; Cantarero, A.; Cebrián-Márquez, G.; Riggio, R. ONIX: Open Radio Network Information eXchange. *IEEE Commun. Mag.* **2021**, *59*, 14–20. [CrossRef]